(12) LEVEL II

RESEARCH IN INFORMATION PROCESSING

Final Report of Research
Performed Under Contract F44620-73-C-0074

DEPARTMENT of COMPUTER SCIENCE

Carnegie-Mellon University

RESEARCH IN INFORMATION PROCESSING

Final Report of Research
Performed Under Contract F44620-73-C-0074

Submitted To:
Defense Advance Research Projects Agency
and
Air Force Office of Scientific Research

Principal Investigators:
Allen Newell
Joeseph Traub

Edited By:
Mark Fox
Sharon Burks

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania
23 December 1978

Table of Contents

1. INTRODUCTION

This is the final report on Research on Information Processing, supported by Defense Advanced Research Projects Agency and monitored by the Air Force Office of Scientific Research under contract number F44620-73-C-0074. It covers the contract period July 1, 1976 to June 30, 1978.

The report is organized as five chapters under five major areas of research: Image Understanding systems, Speech Understanding systems, Machine Intelligence, Software Technology, and Multiprocessors. Each chapter is self-contained and can be read without reading the entire report.

As can be seen from the report, the research efforts cover a wide spectrum of computer science and engineering. One of the strengths of the CMU environment is that the efforts are strongly interdependent and benefit from each other significantly. Efforts on Image Understanding Systems and Speech Understanding Systems benefited significantly from the effective use of Artificial Intelligence techniques on Knowledge Based Systems. The architecture research on Multiple Processor Systems is significantly affected by requirements of Artificial Intelligence and Systems research. Conversely, there is substantial use of Artificial Intelligence techniques in other projects, e.g., Computer-Aided Design and Production Quality Compiler-Compiler. Thus, work in all areas becomes important to the success of any of them.

In the remainder of the report we provide a brief description of research in each of the five areas supported under the present contract. Additional details are available in the reports described in the annotated bibliography at the end of each chapter.

2. IMAGE UNDERSTANDING SYSTEMS (IUS)

Image Understanding Systems (IUS): The major question of interest is: How to interpret visual images by computer within the context of some task environment. The current focus is on high quality multi-spectral images of natural scenes (as produced by high quality colored photographs or by ERTS). Research in this area is expanding in accord with the initiation of IPTO's Image Understanding Systems program.

2.1 Introduction

This section describes the investigation of IUSs capable of producing symbolic interpretations of high-quality multi-spectral images and the construction of appropriate experimental IUSs as a means of testing and exploring the issues.

Specifically, (1) the construction of a low-cost IUS for experimental work; (2) the discovery and development of the body of knowledge to be used by an IUS in interpreting an image; (3) the extension and improvement (both in performance and efficiency) of image segmentation techniques previously studied; (4) the construction and experimentation with an IUS concept demonstration system; (5) the conduction of an experiment jointly with CDC on using symbolic registration for change detection; and (6) adaptation of EBAM technology to the IUS task.

2.2 Background: Pre-Contract Period

Image understanding systems (IUSs) are systems that perceive the external world through visual images. An image is an 2-dimensional array of intensity vectors generated by an external world through some transducer (eg, a retina or a camera). Each intensity vector (called a pixel) may have more than one intensity coordinate (then called multispectral). IUSs operate in conjunction with total systems that use the perceptual input to accomplish some larger task. Thus perception is ultimately evaluated against its role in such performance, ie, whether it permits the system to understand enough to accomplish its intended task.

Research in the area aims at understanding and constructing such systems: (1) understanding the structure of the visual environment that permits perception; (2) discovering the representations, algorithms and overall control structures necessary to exploit the sources of knowledge; and (3) inventing special architectures and programming structures required to realize the algorithms efficiently.

The field is in the midst of a major push on the interpretation of high quality natural scenes. This builds on earlier work on more limited problems (and is itself still a long way from dealing with full perceptual capacities). Work on pattern recognition, in which the image is taken as a single unit to be classified, was the first perceptual task approached and has been extensively investigated. Full analysis of visual scenes with higher sources of knowledge, but in a limited world of straight lines, flat planes, and sharp edges (the world of blocks) was tackled next (and largely with ARPA support in the AI Labs of MIT, SRI and Stanford). This has been extended to some work on highly restricted natural scenes. Work on low quality, black and white natural scenes (such as micrographs of cells) has also received attention. Finally there is a substantial base of work in the image processing of high quality pictures, which has extended to some multispectral classification tasks. Substantial success in all of these set the stage for the IUS research program currently operating by IPTO, which reaches for the next set of requirements along the way to full perceptual ability for machines. The recent work on speech understanding systems (essentially the IPTO SUS program) is also a major component of the current state of the art, since it provides appropriate models for the overall organization of perception systems, namely multiple knowledge sources operating on a common, stratified representation, and the Harpy network coupled with the Beam search algorithm.

In taking this next step in intelligent perception there are three major sources of difficulty. (1) High quality multispectral images are characterized by immense amounts of data (10^7 pixels per image). This

computational size is an inherent feature of image analysis. Solutions to the problems posed by this volume of data must be sought in the structure of the computer system -- devices, architecture, operating software. (2) There is only a small stock of knowledge about the appearance of images generated from the real world. This knowledge is needed at all levels of aggregation. The knowledge deficit is historical. Little scientific effort has been expended in discovering and characterizing all the sources of knowledge that can be brought to bear on the interpretation of an image. Compared with speech, for instance, there has been no long standing study of visual iconics, except for a few special topics (eg, the analysis of perspective). (3) An IUS is a large complex total computational system. Large systems are certainly constructable, but not without major care and expense, especially when the system embodies many untested hypotheses.

The specific long range goal of the CMU effort is to construct complete IUSs: To take as input a high quality multispectral image (say of 2400*3200 pixels) and to interpret it within a task situation in useful time. Such systems imply attention to the full range of issues: architecture, software, algorithms, control structure, heuristics and sources of knowledge. Our research is not funded at a level that makes a coordinated direct attack on this ultimate goal possible. (We do believe such an attack is feasible from a research standpoint, though it would require substantial time as well as funds.) Our actual goals are more modest than this, but they can be understood entirely as steps toward this goal.

Underlying our research goals in IUSs, in concert with our goals in MI, is a general goal of understanding the nature of perception and developing perceptual systems. Perception is by necessity intelligent (ie, adapted towards ends) and we seek to discover the common structure of perceptual systems.

2.3 Relevance of Research

The stage of current work on visual perception -- understanding of high quality multi-spectral natural scenes -- is the first stage in which there is no doubt about the relevance to important applied situations. (Actually, low quality single spectrum natural scenes have an important range of applications, such as automatic karyotyping, but their applicability to the military mission is problematical.) As with most computational applications, the relevance depends sensitively upon the quality of performance obtained, upon the computational costs, and upon the real-time availability.

The paradigmatic tasks being worked on in the IPTO IUS program -- photointerpretation of aerial photographs, guidance of vehicles from images and cartography -- constitute highly relevant task domains. The research carried out was on pictures taken from within this domain, to assure the relevance of the technique and systems developed.

The problem of computational requirements demanded by these applications, required careful attention to the entire system hierarchy (from the devices to the sources of knowledge). Initially a crude analysis (Reddy & Newell, 1977) indicated that very large optimization factors may be possible. The CMU effort, which operates over almost the entire range of system levels, is especially relevant in obtaining such computational goals.

2.4 Accomplishments: Pre-contract Period

Before the 76-78 contract period CMU had a small effort in IUS for several years as an adjunct to the SUS work, where it has provided a second perceptual system for contrast with the SUS. It also served to prepare ourselves to initiate a substantial IUS program, both in thinking through the issues and in bringing ourselves up to speed technically; the present research program is the result. It was already sufficiently far advanced to pose immediate research objectives of major significance to be reached within one and two years.

Significant work had been done. The work of Ron Ohlander (1975) in

developing a method of region splitting using histograms for the segmentation of choice for high quality natural scenes is an important contribution. These results were obtained in the context of a total IUS design, and were simply the first to emerge. Ohlander's thesis also contains significant, though more preliminary, results on the problem of shadows and highlights, and on occlusions.

CMU is one of the few groups concerned with IUSs (as opposed to image processing) that has been working on natural scenes with multi-sensor data. And it appears to be the only group that also works with very large images (eg, ERTS). The CMU Natural Image Data Base is now used by several other research groups (eg, U. Mass, Maryland, USC, SRI).

Before the contract period, work had been done on large images and hence had been computationally extensive. Thus attention had been given to the algorithmic structure of these image techniques (eg, for the Ohlander region splitting technique). Very large reductions (20 fold) in processing requirements were obtained.

Our initial efforts, though small, spanned many of the issues required for approaching the total IUS problems. We created techniques for a number of aspects in addition to those mentioned above: eg, for multi-sensor image extraction, for crude local texture analysis and for change detection.

2.5 Accomplishments: Contract Period 1976-78

2.5.1 Low Cost IUS Experimental System

A system for experimental work with high resolution multi-spectral images was constructed. Its architecture was functionally specialized for image processing. While a proposed performance of 25 Mips was not achieved, the resulting system contained two processors of 5 Mips micro-codable power each. The system was modeled after C.mmp. It has a 4 X 4 cross point switch allowing multiple access to shared memory (fig. 1). The Unix operating system was extensively modified to operate with multiple processors. The

MIPS

HARDWARE ORGANIZATION

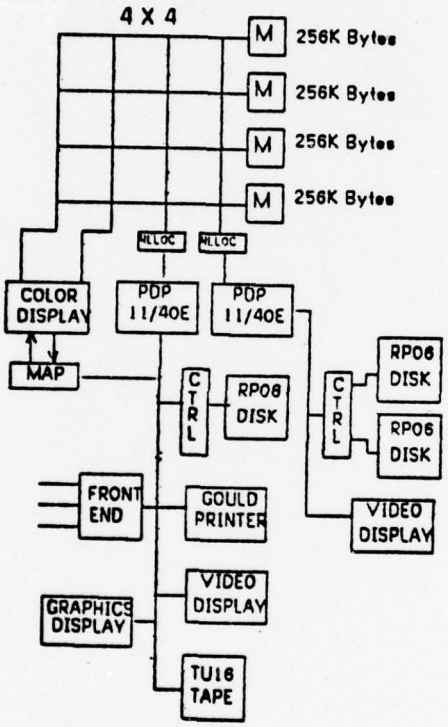

FIGURE 1

system's components were operational by June 1977 and the full system was operational before June 1978. Most image understanding research at CMU is currently done on this system.

2.5.2 Knowledge Acquisition

Research in the acquisition of knowledge used in understanding images was initiated. Knowledge at all levels of the task, features to task structure, were investigated. The primary method of analysis was semi-automated protocol analysis (Akin & Reddy, 1976). The protocols of subjects carrying out image understanding tasks under conditions similar to the computer's were analyzed. Low level features, and higher level abstraction operators were acquired and tested in the experimental system constructed by Keith Price (Price & Reddy, 1977; Price, 1977). A catalogue of knowledge can be found both in Price's thesis and in the article by Akin and Reddy. The extension of protocol analysis techniques to eye movement data was unsuccessful. The goal of this research was to acquire focus of attention data from eye movement. Initial results were found to be uninterpretable, hence the experiments were discontinued.

2.5.3 Improvement of Image Segmentation

The image segmentation techniques that we had (based on Ohlander's work) needed substantial improvement in efficiency and accuracy. The needed extensions were to additional features, especially direct depth measures and multiple texture measures. They also need extension to occlusion, shadows and highlights (along the lines of work initiated already by Ohlander). The next step above segmentation, region classification, began when an adequate processing rate for segmentation was achieved. No good measures existed for evaluating image segmentation; this problem was addressed as well.

The work of Price extended Ohlander's earlier work. Histogram techniques were modified to work on monochromatic images with good results. The earlier histogram techniques required multispectral data in order to achieve satisfactoy segmentation. Price's work was applied to 20 images of widely

varying content. Cityscape, house, landsat, SLR, rural and urban scenes were analyzed. The first three were multispectral input while the last three were monochromatic.

An order of magnitude speedup in segmentation was achieved by Price through the introduction of Planning. An approximate segmentation was generated using a reduced version of the input. The approximation was used to guide the derivation of the final segmentation. The accuracy of segmentation techniques were measured in two ways. First, the performative aspect of segmentation techniques were measured with respect to the task. They were found to be sufficient. Secondly, since the segmentation process of Price's utilized multiple sources of knowledge, an analysis based on decoupling knowledge sources resulted in an estimate of the performance of specific knowldge in a given context.

2.5.4 Concept Demonstration of Image Understanding

Research towards the construction and demonstration of a full image understanding system culminated in the ARGOS system (Rubin, 1978). The Argos task was more complex than originally proposed. Instead of three satellite images, 15 city scenes of downtown Pittsburgh from different views were analyzed. Buildings were labeled with 60% accuracy. The system was demonstrated in January 1978 and is currently being evaluated to improve its accuracy.

The knowledge used to carry out the analysis were area maps, and features and structures of buildings. Knowledge was represented in a network form which extended the Harpy speech system network representation to two dimensions. The search algorithm was also an extension to two dimensional data of the Harpy beam search.

Another important result of this work was the creation of hierarchies of network representations. The results of searching one network restricts the search in the next network. The dynamic creation of networks describing a particular view of the city from a three dimensional network model was also

successfully investigated.

2.5.5 Change-Detection using Symbolic Registration

Experiments in registration and detection of change in image pairs were carried out at the symbolic level by Price (1977). Analysis focussed on size and location differentials in change detection. In symbolic registrations, region pairs were described using different features (knowledge sources) and by abstracting (reducing) image dependent size and location features, enabling their successful matching of region pairs in other images. In both registration and change detection, symbolic analyses proved successful in recognition and prediction.

The data analyzed by Price were supplied by CDC. In conjunction with CDC, registration experiments were conducted. Price's algorithms provided registration information at the symbolic level. CDC's signal level procedures used this information to guide the signal level registration objects. The expected participation by CDC in the signal analyses did not occur. CDC's task was changed to participating in the design and construction of a 60 MIPS signal processor (SPARKS). Many of the key ideas in Spark's architecture were developed during the contract priod. Sparks is expected to be delivered by Fall 1979. It is currently funded independently.

2.6 Annotated Bibliography

This section contains a partial list of reports published in the area of Image Understanding Systems. An abstract accompanies each.

Akin, O., Reddy, R., Ohlander, R. and Schultz, M. Working papers in acquisition of knowledge for image understanding research. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, December, 1976.

Use of knowledge has facilitated complex problem solving in many areas of research. However, in the Image Understanding area, we do not have any systematic treatment and codification of knowldge that is useful in image perception. Further, we do not even have adequate tools for acquiring the necessary knowledge base. In this report we present an experimental paradigm for knowledge acquisition, discuss an analysis technique, and illustrate the different types of knowledge that seem to be useful in image understanding research. In the first paper, three major aspects of knowledge are presented: primitive Feature Extraction Operators, Rewriting Rules, and Flow of Control. A limited number of Feature Extraction Operators were repeatedly used by the subjects to specify location, size, shape, quantity, color, texture, and patterns, of various components found in scenes. Six types of rewriting rules were identified; assertions, negative assertions, context-free, conditional, generative, and analytical inferences. Flow of Control exhibited characteristics of an hypothesize and test paradigm capable of using imprecise, conflicting hypotheses in cooperation with others in a multi-dimensional problem space. The second paper discusses the picture-puzzle paradigm and the various ways in which it can be used as a tool for acquisition of knowledge. The third paper deals with a computer program that assists the transcription of typical protocols obtained from the picture puzzle tasks. Finally, the last paper of the report discusses the pros and cons of using eye-fixation data to acquire knowledge used in some tasks of the picture-puzzle paradigm. The total effort represents an account of the initial results of a new exprimental paradigm. We hope that this will provide a sound basis for understanding the issues of knowledge used in visual perception and aid in the modeling of "seeing" systems.

Hon, R. W. and Reddy, D. R. The effect of computer architecture on algorithm decomposition and performance. In High Speed Computer and Algorithm Organization. (Kuck, D. J., Lawrie, D. H. and Sameh, A. H., Ed.) Academic Press, New York, NY, 1977, pp. 411-422.

There are many problems in image processing and signal processing which require processing powers in the range of 10*8* to 10*10* instructions per second. Cost-effective solutions to these problems require some form of functional specialization in the computer architecture. The processor- memory-switch structure of the functionally specialized system, in turn, affects algorithm decomposition and overall performance. Depending on the design choices, different aspects of the system become the main bottlenecks. In this paper we examine the effects of processor speed, memory access time, bandwidth, and capacity on algorithm decomposition and system performance.

Kanade, T. Model representation and control structures in image understanding. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

This paper overviews and discusses model representations and control structures in image understanding. Hierarchies are observed in the levels of description used in image understanding along a few dimensions: processing unit, detail, composition and scene/view distinction. Emphasis is placed on the importance of explicitly handling the hierarchies both in representing knowledge and in using it. A scheme of "knowledge block" representation which is structured along the processing-unit hierarchy is also presented.

Kender, J. R. Saturation, hue and normalized color: Calculation, digitization effects and use. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, November, 1976.

Three transformatons used in the analysis of tricolor natural scenes are analyzed. All have nonremovable singularities, near which they are highly unstable. Given digital input, the distribution of their transformed values is highly nonuniform, characterized by spurious modes and gaps. These effects are quantified and illustrated. In addition, a significantly faster algorithm for hue is derived. Image segmentation techniques of edge detection, region growing, clustering, and region splitting are affected arbitrarily badly by such problems. Some stratagems are illustrated that help minimize the bad behavior. Linear transformations are presented as a generally favorable alternative to these three nonlinear ones.

Kriz, S., Reddy, D. R., Rosen, B., Rubin, S. and Saunders, S.

Academic Press, Pittsburgh, PA, 1978, pp. 39-62 set, and performance of H Harp..HARP: A low-cost 25 MIPS digital procesor. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, February, 1978.

Harp is an ultra-high performance processor motivated by the low-level processing requirements of maching perception tasks. It is designed for extreme speed, low cost, easy producibility, simplicity of structure and programming, and complete diagnosability. This paper discusses the design philosophy, architecture, instruction

McKeown, D. M. and Reddy, D. R. A heirarchical symbolic representation for an image database. Proceedings of the IEEE Workshop on Picture Data Description and Management, Chicago, IL, April, 1977.

This paper describes the structure and facilities available in MIDAS, a Muli-sensor Image Database System, currently under development at Carnegie-Mellon University. We expct that MIDAS will be used as a tool by researchers for image analysis and description studies. The facilities in the database have been motivated by the needs of various researchers in our group. MIDAS is designed for use as an aid in performing the following types of tasks: performance evaluation, error analysis, and knowledge acquisition. Unlike other image database systems, the pimary data structure chosen for scene representation is a hierarchic symbolic description. However, other representations such as a relational database are also maintained for fixed specific needs. This report discusses several design choices which led to the current organization of MIDAS. The features of the MIDAS system are designed to be machine independent. The particular implementation discussed is on a multi-processor PDP-11 system with large file and memory capabilities using the UNIX operating system.

Price, K. and Reddy, D. R. Change detection and analysis in multispectral images. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977. Also CMU-CSD Technical Report December, 1976.

This paper describes work on the development of symbolic regis- tration and change analysis techniques applied to the problem of the comparison of pairs of images of a scene to generate descriptions of the changes in the scene. Unlike earlier work in change analysis, all the matching and later analysis is performed symbolically. We also discuss techniques for the generation of symbolic descriptions of images, both the segmentation and feature

extraction problems. These techniques have been applied to several different scenes in in this paper we present the results for two of these scenes.

Rubin, S. M. and Reddy, D. R. The locus model of search and its use in image interpretation. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

The central problem in image understanding is the representation and use of all the available sources of knowledge in the interpretation and description of an image. The problem of representation is complicated by the diversity of sources of knowledge. Converting knowledge into effective algorithms in the presence of error and uncertainty further complicates the issue. In this paper we present a specific framework for representation and use of knowledge which appears to be both sufficient and efficient for a wide variety of image interpretation tasks. The framework for image interpretation presented here is based on the Locus model successfully used in speech understanding research (Lowerre and Reddy, 1977). The Locus model is a non-backtracking, non-iterative, deterministic search technique in which a beam of near-miss alternatives around the best path are extended to determine the near-optimal descrip- tion of the image. In the following sections we will outline the structure of the model and discuss the relationship of the present approach to earlier attempts at image interpretation. A complete version of this paper, including a detailed example has been submitted to IJCAI-77 and can be obtained by writing to the authors. A detailed description of the model as applied to image interpretation task will be given in Rubin (1977). A more complete discussion of the strengths and limitations of the model and its relationship to the other approaches to knowledge representation and search are given in Reddy (1977).

3. SPEECH UNDERSTANDING SYSTEMS (SUS)

Research on SUS at CMU has been conducted as part of the larger 5-year SUR program of DARPA. This entire program was terminated by DARPA in October, 1976 at the end of the 5 year effort, i.e., 3 months after the start of the current contract period.

As part of this program, CMU developed two systems, HARPY and Hearsay-II, which were two of the three 1000 word systems demonstrated at the end of the program. HARPY was the only system to meet the original specifications and in fact exceeded several of the requirements.

A complete Final Report summarizing the results of research over the entire five year period was published in September, 1976 and reprinted in August, 1977. The reader is referred to that report for complete details.

4. MACHINE INTELLIGENCE (MI)

Machine Intelligence (MI): The major question of interest is: How to create computer systems capable of intelligent action? The focus is on basic control structures (Blackboard control, derived from HEARSAY-II, and production systems), heuristic search and knowledge-intensive problem solving. Research in this area is moving toward a major test of a production system.

4.1 Introduction

This section describes our exploration into a range of basic mechanisms and organizations with respect to constructing computer systems capable of intelligent action.

Specifically: (1) the construction of an "instructable production system" as a means of assessing productions systems as a system organization; (2) the exploration and evaluation of a multiprocessor version of the Blackboard Model of system organization; (3) the continuation of our study of heuristic search techniques, by both theoretical and experimental techniques; and (4) the continuation of our study of how to represent the knowledge in scientific-technical domains so it can be used for machine problem solving.

4.2 Background: Pre-Contract Period

Since machine intelligence (MI) is the study of how to obtain intelligent action by computers, it must encompass both the attempt to discover the principles whereby intelligent action is possible and to construct computer systems that can perform tasks requiring intelligence.

Research in machine intelligence covers a wide range of issues: (1) discovering and analyzing methods of problem solving; (2) discovering and analyzing the way problems are represented, and how that representation affects the difficulty of solving problems; (3) discovering and analyzing the processes that recognize situations and describe them, thus obtaining appropriate internal representations; and (4) discovering and understanding the control structures and system organizations that are used to put together a collection of problem solving methods and representations into an effective

total systems. The domain of motor activity and its control, while logically a part of MI, has not yet reached parity intellectually with the others.

Some of these issues can be studied analytically (eg, the analysis of a search algorithm); others require constructing experimental systems. The body of scientific knowledge that emerges provides the basis on which we (mankind) will create ever more sophisticated computers, capable of solving difficult problems for us, and of taking care of the full range of errors, contingencies and variability that haunt all naturally occurring problems.

The field of MI (or artificial intelligence) has existed as a serious scientific endeavor only since the mid 1950s. Within that time it has come to be accepted as a major subfield of Computer Science (a scientific field that is not much older). A number of basic results of both generality and power have been obtained. These permit the transmission of know-how sufficient to produce machines that exhibit intelligent behavior within those relatively limited situations which can be described symbolically with some economy. A short qualitative statement of some of the more general of these results can be found in the Artificial Intelligence Section of the 1975-76 CMU Arpa Proposal.

The existing body of knowledge and technique is sufficient to produce applications when the circumstances are extremely favorable (as with Dendral, Hearsay-II, and Harpy), but not as a routine matter. The field is simply in mid-flight with respect to attacks on its basic problems. In almost every aspect -- search, representation, description, control -- we have explored second, third and even fourth generation mechanisms and systems. But in only a few places has stability in our scientific understanding yet developed. Fundamental investigation is profitable in all areas.

One way to describe the essential problem is that we are here touching one of the great scientific mysteries -- the nature of intelligence. We are approaching it through a particular avenue, constructing intelligent machines, and sampling the whole at only a few places. It will be some time before

enough of the picture accumulates to really see what is essential. There are whole stretches of intelligent action that have never been attended to in any scientific detail. Related to this is that the major source of knowledge about intelligent action is man himself and how he thinks.

Another way to describe the essential difficulty is that the nature of symbolic reasoning is such that it is easy for one mechanism to masquerade as another (the essence of simulation). It is possible to take an approach that has a grain of truth and force it to yield additional results, though with increasing difficulty. Alternative candidate mechanisms for intelligence can coexist for quite a while before resolution occurs, for each can be made to accomplish some of the functions of the other. There is some indication that this is the situation with respect to control structures and system organization, in which there are now several candidates which are becoming rather well developed.

CMU has been involved in all of the major areas of development of intelligence for machines throughout the 20 year history of the field, being one of the three major university centers for work in artificial intelligence in the US (along with MIT and Stanford). Our long term goal is thus not less than the complete scientific understanding of the nature of intelligence. The operative definition of "understanding" in this context is to produce systems which exhibit recognizably intelligent behavior over a large spectrum of tasks. (Our interests also extend beyond machine intelligence to the nature of intelligent action in humans -- though that is not the center focus of the IPTO work.) Our short run goals have changed over the years as the field has developed and our knowledge matured.

In our IPTO effort we place a very large emphasis on perception. This is covered by our work described under the SUS and IUS projects, but it is to be understood as a major part of our basic orientation in machine intelligence.

Besides perception, the main goal is to discover the appropriate basic underlying control structure and system organization to be used in an

intelligent agent. There are several candidates being explored in the field at large (eg, Planner-like systems, semantic nets, heterarchical systems). We explored two schemes intensively: the Blackboard Model and Production Systems. One derives from our own work in SUS, as described in section 3.2. The other derives from the study of how humans solve problems and what basic organization they seem to have. The two control models are related, but have a number of important differences, and thus represent genuinely different alternative structures.

As in our other work (eg, Hearsay vs Harpy), we found it useful to investigate more than one alternative structure. The net result is not simply the selection of one or the other, but the emergence of a control structure that is the synthesis of the two. (This happened in fact between Hearsay and Harpy, which seemed initially to embody completely opposite philosophies).

We maintained a significant effort in understanding the nature of heuristic search. Search is an absolutely fundamental characteristic of intelligent action. Whenever there is uncertainty (and what is more characteristic of situations requiring intelligence?) a fundamental response is to posit what might be so and then proceed to discover the consequences. Discovery of negative consequences leads to abandoning the posit and trying another, ie, to search. Several uncertain aspects lead to several cascaded posits, which creates thereby combinatorial search.

Analysis of search is of great importance to all schemes for producing intelligent action, and to all tasks areas in which intelligence is to be applied. For instance, it is vital to the work on complex perception (the IUS and SUS efforts), though to naive judgment perception seems far removed from search. Currently, other problems occupy center stage for most of the MI field, eg, problems of representation of common sense knowledge; CMU has one of the few efforts directed at understanding search.

The ultimate goal of work in search is to discover and understand how knowledge can be brought to bear to control the search. The CMU goal has one

part that is attempting to discover new mechanisms of search control and a second part that is attempting to create models of the search process that permit a mathematical analysis of search strategies (ie, to prove that certain strategies are optimal under certain conditions).

We also have some research goals for the study of the representation of knowledge. The general problem is that of discovering what knowledge in various task domains is useful to problem solving, and then discovering how to represent it so that it can become available to problem solving methods. We have alluded to this problem in both IUS and SUS, but especially in IUS where acquiring knowledge about visual cues is an essential part of current research. In much of MI, currently, the problem of representing knowledge of the everyday world is cast in terms of its use in understanding natural language input. The CMU goal is to understand the representation of knowledge in technical and scientific domains, eg, what knowledge does a chemical engineer actually have, and how is it organized so he can solve those problems that require his chemical knowledge. Such domains are quite different from either the perceptual domains or those of everyday life. Little work is going on elsewhere with respect to such task domains.

4.3 Relevance of Research

The aims of research in MI are so broad -- nothing less than extending the capabilities of symbol processing machines for intelligent action -- that it is not always easy to identify applications as emanating specifically from research in this field.

Consider the important contributions that MI has already made. List processing has become an integral part of programming technology. The abstract theory of programming owes much of its early impetus to Lisp. The initiation of verification of programs is likewise due strongly to work in machine intelligence. This is also true of the subfield of symbolic mathematics. Even some of the concepts of structured programming can be traced back to early MI languages -- concepts of extensive hierarchization and

recursion, for instance. Similarly, fundamental ideas of heuristic search are used widely in operations research programs for domains where powerful optimization techniques are unavailable or inadequate. Heuristic search is especially necessary today for handling large combinatorial problems, such as job-shop scheduling.

As these examples illustrate, MI applications are characterized by helping to initiate fields of application and then becoming freely mixed with independent invention and development from within the field. For example, in operations research, branch-and-bound techniques for limiting search (analogous to alpha-beta procedures in game trees) and optimal scheduling algorithms came not from MI, but from within operations research. Structured programming and symbolic mathematics have run essentially independent courses from work in MI. An extreme example of this "initiation" syndrome in MI applications was the strong effect of MI on the initiation of time sharing, but with little specific technical transfer.

The reported MI research has some ties to specific areas of application, as we note below. However, its ultimate fate is likely to be similar to the examples above, in which most of the applications will not be identified as MI. For instance, work on control structures is of fundamental significance to future applications. Every intelligent system must employ a control structure capable of using partial knowledge, discovering relevant knowledge, coping with pervasive error, etc. But as we discover effective system organizations, they will become assimilated into the application area, their further development being seen as part of that application. Similarly, progress in heuristic search, being of general utility, will diffuse through various applications fields.

The particular efforts in this report do have some specific claims to being relevant for particular application areas.

Production systems already give strong indication of becoming the appropriate system organization for a class of applications which can be

characterized as user-oriented interactive knowledge experts. Currently, the prototypical program is MYCIN, which gives advice on antibiotic drugs. The work in production systems at CMU was an initiating source for these applications, though they have developed strongly in their own way as well (providing another example of the role discussed above). Our current efforts in production systems lead to a better understanding of this control structure, providing direct support for this application thrust. These efforts also lead to significant expansion of the control structure, in particular in showing how to grow production systems from the outside, ie, without having detailed knowledge of the internal structure of the knowledge already encoded into the system. This is a step towards systems which can gain their knowledge through interaction with non-experts and from an inanimate (more precisely, non-instructing) environment.

The Blackboard control structure comes directly out of the work on speech understanding. SUS (and IUS) are themselves simultaneously areas of MI and areas of application, as discussed in their own sections. Thus, the primary relevance of the application of the Blackboard Model is covered in the SUS (and IUS) sections. The relevance of the work on C.mmp lies primarily in the prospect that the cost-effective implementations of SUSs and IUSs will be in multiprocessor configurations, due to the development of LSI technology. The Blackboard Model is well suited to multiprocessor realization, but there is much to be found out to make such implementations effective. The issues are not programming issues in the usual sense, but issues of decomposition and coordination (and how much various solutions cost).

The work on new control mechanisms in Heuristic Search is relevant to the entire range of areas where intelligent systems are built. The specific mechanisms under investigation in our work are members of an important new class of mechanisms. Heretofore one took the very general algorithm for heuristic search and added to it highly task specific heuristics to obtain selectivity (eg, evaluation functions). Alpha-beta was one of the few

mechanism of both generality and power. The causality mechanism and the invariance mechanism are both like alpha-beta in that they are both powerful but general, so that they transcend the particular task domain. Thus, they have a strong claim on being relevant for future applications.

Even in our own environment, heuristic search is an important component of several other areas. SUSs and IUSs are by now obvious. But heuristic search shows up in both the Production Quality Compiler Compiler and in the RT-level Computer Aided Design System.

Incorporating (into intelligent systems) the real-world knowledge that an intelligent and professional human problem solver applies to his tasks provides a major bridge in moving from problems of laboratory interest to problems of real-world importance. The work on representation of knowledge enables designers of intelligent systems to augment general problem solving techniques with specific information to make problem solving more efficient in a specific task domain. Evidence has accumulated that the proficiency of professionals rests at least as much on such specific information as upon any superiority in their problem solving skills, or their general knowledge of the world (common sense). The focus of CMU's work here is especially important since much of the rest of the field is attending to the representation of common-sense knowledge.

In summary, the research in MI is aimed less at designing specific intelligent systems than at deepening our understanding of the principles of organization underlying successful intelligent systems of all kinds. Hence, the results of the research are relevant to the work of systems designers over a wide range of task domains. When areas of application of MI do show up, even in our own shop, they tend to get defined under labels other than MI, ie, the work in SUS and IUS.

4.4 Accomplishments: Pre-Contract Period

CMU has been a major contributer to MI for twenty years (since 1956). Although much is ancient history, the following brief summary of highlights might be useful. The early work (to about 1961) was all done jointly with Rand. We leave to one side all the work on IUS and SUS, which has been covered in other sections. We also do not touch on the psychological work unless it also has implications for work in machine intelligence. (Since much of the work extends over several years with several investigators, the dates are only approximate, usually a first publication, and the names are included only when needed for identification.)

- Developed the first heuristic search program (LT, the Logic Theorist, 1956).
- Discovered list processing and developed the first list
- processing languages (IPL2 - IPL5, 1957).
- Put forth the growing discrimination net as the theory of man as a symbolic information processing system (1958).
- Developed the first general problem solving program (GPS, 1959) which embodied four major ideas:
    - Means-ends analysis, a major problem solving method.
    - Goals as general data structures to guide problem solving activity.
    - Independence of the problem solving methods from the specific task structure.
    - Planning as a method of problem solving involving creating an abstract problem space and developing in it a guide for detailed action.

  Put forth a theory of human memory organization for simple verbal learning (EPAM, 1959-60).
- Developed the first applications of MI techniques to the management sciences (1959-63).
    - Assembly line balancing (1959)
    - Portfolio selection (1962)
    - Production scheduling (1963)
    - Warehouse location (1965)

- Did the first detailed simulations of human cognitive processes (with GPS, 1961).

- Developed the analysis of human protocols, returning them to the status of serious experimental data (1961)

- Formulated a theory of induction in series extrapolation tasks (1962).

- Discovered a search scheme that enabled search to go extrememly deep in pursuit of chess combinations (1965).

- Developed Production Systems as a form of control structure for human problem solving (1966)

- Did the major study that opened up semantic nets as a subfield (Quillian, 1968).

    - [There were precursors: Lindsay (CMU, 1959), Green et al (Lincoln Labs, 1959), Raphael and Bobrow (MIT, both 1964).]

  Formulated a theory of MI as the study of weak methods and characterized the basic methods that appear to compose most intelligent programs (1969).

- Developed the first non-deterministic problem solver that took algorithms expressed in a non-deterministic language and created a constraint satisfaction problem to solve them (REF-ARF, 1970)

    - This program also has substantial generality.

  Developed the first program verification system (King, 1970).

    - This is based on theoretical work done at CMU in 1967 by Floyd in developing the now-called Assertion Method for program verification.

  Investigated the phenomena of superior perception of chess positions by chess masters, extending the theory, and verifying it both by simulations and by experimental data (1971).

    - The discovery of the phenomena is due to DeGroot (1944). It lead to a general theory of what mechanisms underly expertise in a task domain.

  Developed an application of MI techniques for doing automatic analysis of human problem solving protocols and showed feasibility (1971).

- Developed an application of heuristic search techniques to layout of physical space, eg, a computer room (1971).

- Developed linear and merge techniques in automatic theorem proving (1971).

- Put forth an integrated version of the information processing theory of problem solving supported by detailed analysis of data (Newell & Simon, Human Problem Solving, 1972).

- Included the notions of a Problem Space and the Problem Behavior Graph as ways of representing human problem solving behavior.

Developed a uniform theory of the nature of pattern induction and rule learning (1973)

- This includes a generalized method of rule induction.

Developed the formulation of the production system as a model of the human's basic information processing architecture (1973).

- Developed a theory of visualization (the mind's eye) in humans based mostly on production systems (1974).

- Initiated the detailed study of mental development in information processing terms using production systems (1974).

- Developed a theory of how new tasks are acquired (assimilated) from the task environment and a problem space created (1974).

- Developed the concept of a mapping as a representation and the act of mapping as a generalized inferential operation (Merlin, 1974).

The list above is intended to bring us up to the immediate precursors of the work in progress and proposed. (It should be said of any such list of claims of accomplishments that it does some violence both to the continuity of work at the place being described (CMU) and to the dependence on concurrent scientific work and results of other scientists.)

As is evident, CMU's involvement in production systems goes back to its origins as an organizing scheme for MI. (The earlier work on production systems was in logic and grammar). This work started as a theory of how humans solve problems. It has influenced the work at Stanford (around E. Feigenbaum) in which productions have been used in Dendral, then in some early learning studies, then in Mycin, and now in many efforts (and has spread to Rand's intelligent terminal project). This development has taken a somewhat different form than the CMU work, being more focussed on backward chaining through production rules than on forward generated behavior (as in a machine architecture).

We have been studying production systems as a general control structure for MI for some time. We have created several production system languages (PSG

and PSNLST being the main ones) in exploring the basic architectural structure. We have written about 20 substantial production systems that cover a wide range of MI tasks (systems that range around 200 productions each). Many of these are alternative versions of important MI programs (eg, GPS), so we can evaluate the role that encoding processes as production systems provide. The task environments cover problem solving, vision and visualization, parsing, programming, and problem formulation. We have developed production systems that grow, ie, that add productions in the course of their operation, and we now use learning mechanism routinely in our production systems. Many of these things have not yet been widely disseminated.

In addressing the problem of search we have recently discovered a number of search control mechanisms add fundamentally to the stock of such mechanisms available. For example, we have developed a scheme for assessing the causes of a negative action and use this causality analysis to develop appropriate actions that deal with it. We have recently developed a model of an environment with distributed goals states and with only probabilistic knowledge from any point about where goals states might be. Within this model we have been able to discover and prove optimal search algorithms.

The work in the area of representing knowledge in scientific-technical domains has already resulted in a system for problem solving in chemical thermodynamics. The system is amazingly lean and the ideas appear to be applicable to other domains where there is a strong underlying scientific theory.

4.5 Accomplishments: Contract Period 1976-78

4.5.1 Instructable Production System (IPS)

Research in IPS focused on two tasks. The first is the "Job Shop" task. It concerns the allocation and scheduling of resources and machines in a job shop. The second task, which we call the "Physics" task, solves college level physics textbook problems.

The completion of these tasks was delayed by the slowness of the production interpreter. One of the factors in the success of the Harpy & Hearsay II speech understanding systems was its speed in recognition. The running and analysis of hundreds of utterances allowed us to understand and debug the knowledge necessary to understand speech. Hence, research focussed on solving the production recognition problem and conflict resolution strategies. New techniques for structuring the recognition process and resolving conflict were discovered. This resulted in a speed up by a factor of 80 in production recognition (McDermott et al, 1976) (Forgy, 1977). By the end of the contract period, the production interpreter could execute 10 actions/sec. It is now possible to run systems with over 1000 productions, and work is now proceeding on the "Physics" task and a new task, "Hardware Configuration".

The problem of instructability was also found to be more difficult than anticipated. Research in how production knowledge should be structured to allow the proper integration of new knowledge is currently under way.

4.5.2 Blackboard Control Structure on Multiprocessors

A version of the Hearsay-II architecture was designed and partially implemented on a multiprocessor. The goal of ths work was to undersand the multiprocessing issues of this structure. A skeleton architecture was constructed with simulated knowledge sources. By tailoring the locking primitives, significant processing speed-ups were achieved. Research on a distributed Hearsay-II was also initiated. It was found that even when data locking was removed, Hearsay still converged due to its robust error recovery capabilities.

McCracken (1977) created a production system version of Hearsay. It investigated the decomposition of knowledge into productions, and the potential parallelism achievable. It was found that the production sysem representation afforded a large degree of parallelism coupled with a viable decomposition of knowledge.

4.5.3 Techniques of Heuristic Search

Heuristic search pervades all sectors of MI. Whether in game playing or learning, search is a primary mechanism, and methods that control its exponential explosion are crucial to MI. During the contract period, a multi-faceted attack was made on the problem.

Berliner's (1977) work in game playing introduced the concept of extracting invariances. An invariance describes (i.e. what was learned there) a portion of the search tree and is used to guide search in other areas. A second concept introduced by Berliner is "Theme Broadcasting". By broadcasting the purpose for searching a particular portion of the game tree, a node's evaluation function can be made context-sensitive.

In an attempt to better understand how evaluation functions affect search, Berliner (1977) introduced the concept of multiple evaluation functions dependent upon state-class, hence, achieving better performance through localization of knowledge.

Berliner (1978) discovered a new search algorithm, B*. It uses both upper and lower bounds of a node evaluation to carry out selection. Greater emphasis is placed on knowledge used in node evaluation. It is shown that B* is more effective in searching adversary trees than current methods.

Fox and Reddy (1977) investigated the types of knowledge necessary in reducing the search space in the learning of word pronunciation dictionaries for speech understanding systems. Four types of knowledge were recognized and applied. Search (learning) time was significantly reduced.

Gashnig has extensively studied the theory of heuristics. In Gashnig (1977a), the effect of redundant computation in search was analysed. A new backtracking algorithm was discovered that greatly reduced search time by a space - time tradeoff.

Gashnig has also elaborated a theory for measuring the relative efficiency of heuristics. A number of measures were empiracally derived that

characterize the search spaces of heuristics. An analysis was also made on the effects of combining heuristics linearly (Gashnig, 1977b).

4.5.4 Representation of Knowledge

Constructing expert problem solving programs for scientific and technical domains poses two problems. 1) How to represent the knowledge necessary to solve problems and 2) How to bring this knowledge to bear. Two task domains have been investigated.

The work of Bhaskar and Simon (1977) investigates how experts solve problems in chemical thermodynamics. A theory of problem solving in the thermodynamics domain was constructed, and embodied in an interactive protocol analysis program. Means-End analysis is the primary problem-solving technique of the theory. Deviations from the theory were shown to occur only under conditions of error.

Fox and Mostow (1977) investigated the problem of constructing a consistent interpretation of errorful data in the domain of speech understanding. A "Conceptual Hierarchy" was created as a representation of the domain's semantics. Data to be interpreted (eg. sentence fragments, ungrammatical sentences) were assimilated into the hierarchy, followed by a search in the hierarchy for a consistent interpretation. Applications of these techniques are forseen in vision and concept recognition.

Fox and Reddy (1977) describe the types of knowledge needed to learn word pronunciation. They show how this knowledge can be brought to bear in reducing combinatorial search in the learning algorithm.

4.6 Annotated Bibliography

This section contains a partial list of reports published in the area of Machine Intelligence. An abstract accompanies each.

Berliner, H.

This paper proposes a set of new techniques for tree searching. Existing methods make use of domain-dependent information only when calculating evaluation functions. The scalar value that is the result of such an evaluation is then used for all future decisions that control the search. Our technique provides a method for using domain-dependent descriptions to rule out the searching of subtrees, based upon notions such as causality, purpose, consistency, and problem invariance. Descriptions are inherently much richer than scalar quantities, and we develop a theoretical framework for this. The techniques discussed were developed using computer chess as a domain, but they appear to be widely applicable.

Berliner, H. Experiences in evaluation with BKG--A program that plays backgammon. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977. Also CMU-CSD Technical Report July, 1977.

Because of very high branching factors, a backgammon program must rely on knowledge rather than search for performance. We here discuss insights gained about the structure of evaluation functions for a large domain such as backgammon. Evaluation began as a single linear polynomial of backgammon features. Later, we introduced state- classes, each with its own evaluation function. This improved the play, but caused problems with edge-effects between state-classes. Our latest effort uses models of position potential to select across the set of best members of each represented state-class. This has produced a significant jump in performance of BKG. Because of the localization of knowledge, state-classes permit relatively easy modification of knowledge used in evaluation. They also permit the building of opponent models based upon what evidence shows the opponent knows in each state-class. Our program plays a generally competent game at an intermediate level of skill. It correctly solves a high percentage of intermediate level problems in books.

Berliner, H. The B* tree search algorithm: A best-first proof procedure. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, March, 1978.

In this paper we present a new algorithm for searching trees. The algorithm, which we have named B*, finds a proof that an arc at the root of a search tree is better than any other. It does this by attempting to find both the best arc at the root and the simplest proof, in best-first fashion. This strategy determines the order of node expansion. Any node that is expanded is assigned two values: an upper (or optimistic) bound and a lower (or pessimistic) bound. During the course of a search, these bounds at a node tend to converge, producing natural termination of the search. As long as all nodal bounds in a sub-tree are valid, B* will select the best arc at the root of that sub-tree. We present experimental and analytic evidence that B* is much more effective than present methods of searching adversary trees. The B* method assigns a greater responsibility for guiding the search to the evaluation functions that compute the bounds than has been done before. In this way knowledge, rather than a set of arbitrary predefined limits can be used to terminate the search itself. It is interesting to not that the evaluation functions may measure any properties of the domain, thus resulting in selecting the arc that leads to the greatest quantity of whatever is being measured. We conjecture that this method is that used by chess masters in analyzing chess trees.

Bhaskar, R. and Simon, H. A. Problem solving in semantically rich domains: An example from engineering thermodynamics. Cognitive Science 1, 2 (April 1977).

Recent research on human problem solving has largely focused on laboratory tasks that do not demand from the subject much prior, task-related information. This study seeks to extend the theory of human problem solving to semantically richer domains that are characteristic of professional problem solving. We discuss the behavior of a single subject solving problems in chemical engineering thermodynamics. We use as a protocol-encoding device a computer program called SAPA which also doubles as a theory of the subject's problem-solving behavior. The subject made extensive use of means-ends analysis, similar to that observed in semantically less rich domains, supplemented by recognition mechanisms for accessing information in semantic memory.

Eastman, C. M. and Henrion, M. GLIDE: A language for design information systems. Proceedings of the ACM SIGGRAPH Conference, New York, NY, July, 1977. Also CMU-CSD Technical Report August, 1976.

An integrated database for design is one which incorporates all information describing a proposed entity in sufficient detail for design and construction. One of several computer database facilities with this goal is BDS (Building Description System). It organizes informa- tion as a collection of Elements, each defined as a set of spatial and other attributes. A design data-base should facilitate: Automatic reformatting of data for analysis programs, automatic drafting of production documents, convenient man-machine interaction for input and manipulation of information, easy definition of routines for automatic parts selection and detailing, and flexible data-structures to respond to the constraints of a particular building system or method. This paper defines a language called GLIDE, Graphical Language for Interactive Design, which is intended to meet these needs. GLIDE includes of a basic set of commands for direct operation on the database in interpretive mode. These form part of the complete language which can be used to extend the basic commands by defining new structures and pre-compiled routines. The syntax is Algol-like with the addition of Topology, Element, Set and View records types and structures for complex Shapes. It includes graphics facilities and spatial search operations. This paper is introduced by a discussion of the requirements of such a language, and contains a detailed specification of the semantics and syntax of GLIDE with examples of programs.

Erman, L. D. and Lesser, V. R. The Hearsay-II speech understanding system: A tutorial. In Trends in Speech Recognition. (Lea, W. A., Ed.) Academic Press, Hillsdale, NJ, 1978. Also CMU-CSD Technical Report May, 1978.

The Hearsay-II system, developed at CMU as part of the five-year ARPA speech-understanding project, was successfully demonstrated at the end of that project in September, 1976. This report reprints two Hearsay-II papers which describe and discuss that version of the system.

Erman, L. D. and Lesser, V. R. System engineering techniques for artificial intelligence systems. In Computer Vision Systems. (Hanson, A. and Riseman, E., Ed.) Academic Press, New York, NY, 1978. Also CMU-CSD Technical Report December, 1977.

It is impossible to develop a large knowledge-based artificial intelligence system successfully without careful attention to issues of system engineering. A set of principles is presented for organizing the design and implementation of such a system. Problems of maintainabiltiy and configuration control, human

engineering, performance analysis, and efficiency must be faces. Tools used to solve these problems are described, along with examples of their use in the Hearsay-II speech understanding system. This is a slightly revised version of a chapter to appear in A. Hanson and E. Riseman (eds.), Computer Vision Systems, Academic Press, 1978.

Forgy, C. L. A production system monitor for parallel computers. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, February, 1977.

Production systems cannot compete on an equal basis with conventional programming languages until the efficiency of their monitors is improved. The process that finds the true productions on every cycle is most in need of improvement; even in today's small systems this process is often expensive and it is likely to become more expensive if production systems become larger. There are a number of possible ways to achieve the greater efficiency, including taking advantage of the slow rate of change of the data base, expending a minimum of effort on false antecedent conditions, avoiding whenever possible the operations that are likely to be most time consuming, and making efficient use of hardware. Since computer power is achieved most economically today through parallelism, no algorithm can be truly efficient in its use of hardware unless it can be executed in parallel. A production system monitor has been implemented that responds in a reasonable manner to both the peculiar nature o the task and the realities of current hardware technology.

Forgy, C. L. and McDermott, J. OPS, a domain-independent production system language. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

It has been claimed that production systems have several advantages over other representational schemes. These include the potential for general self-augmentation (i.e., learning of new behavior) and the ability to function in complex environments. The production system language, OPS, was implemented to test these claims. In this paper we explore some of the issues that bear on the design of production system languages and try to show the adequacy of OPS for its intended purpose.

Fox, M. A. and Reddy, D. R. Knowledge-guided learning of structural descriptions. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

We demonstrate how the use of domain dependent knowledge can reduce the combinatorics of learning structural descriptions, using as an example the creation of alternative pronunciations from examples of spoken words. Briefly, certain learning problems (Winston, 1970; Fox & Hayes-Roth, 1976) can be solved by presenting to a learning program exemplars (training data) representative of a class. The program constructs a characteristic representation (CR) of the class that best fits the training data. Learning can be viewed as search in the space of representations. Applied to complex domains the search is highly combinatorial due to the: 1) Number of alternative CRs. 2) Size of training set. 3) Size of the exemplars.

Fox, M. S. and Mostow, D. J. Maximal consistant interpretations of errorful data in hierarchically modelled domains. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

A method is presented for constructing maximal consistent interpretations of errorful data. The method apears applicable to many tasks (speech understanding, natural language understanding, vision, medical diagnosis) requiring partial-matching of errorful data against complex, hierarchically defined patterns. The data is represented as symbolic structures (word sequences, line segment configurations, disease symptoms0. Errors consist of missing data (*unrecognized words*, occluded lines, undetected symptoms) and extra (possible inconsistent) data (incorrectly recognized words, visual noise, spurious symptoms0. Data interpretations correspond to substructures of a hierarchy of predefined concepts. Constraints on consistent conceptual structures are embedded in the hierarchy. An implementation of the method has correctly interpreted errorful sets of sentence fragments recognized by the HEARSAY-II speech understanding system. The implementation has also correctly interpreted typed-in ungrammatical sentences. Detailed examples illustrate operation of the method on real data.

Gaschnig, J. Exactly how good are heuristics: Toward a realistic predictive theory of best-first search. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

We seek here to determine the exact quantitative dependence of performance of best-first search (i.e., A* algorithm) on the amount of error in the heuristic function's estimates of distance to the goal. Comparative performance measurements for three families of heuristics for the 8-

puzzle suggest general conjectures that may also hold for more complex best-first search systems. As an example, the conjectures are applied to the coding phase of the PSI program synthesis system. A new worst case cost analysis of uniform trees reveals an exceedingly simple general formula relating to cost to relative error. The analytic model is realistic enough to permit reasonably accurate performance predictions for an 8-puzzle heuristic. The analytic results also sharpen the distinction between "knowledge itself" and the "knowledge engine itself".

Gaschnig, J. A general backtrack algorithm that eliminates most redundant tests. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

We define a faster algorithm functionally equivalent to the classical backtrack algorithm for assignment problems, of which the Eight Queens puzzle is an elementary example [Fillmore and Williamson 1974, Knuth 1975]. Experimental measurements reveal reduction by a factor of 2.5 for the 8-queens puzzle (factor of 8.7 for 16 queens) in T, the number of pair-tests performed before finding a solution (i.e., first solution).

Gillogly, J. J. Performance analysis of the technology chess program. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, March, 1978.

Many Artificial Intelligence programs exhibit behavior that can be meaningfully compared against the performance of another system, e.g. against a human. A satisfactory understanding of such a program must include an analysis of the behavior of the program and the reasons for that behavior in terms of the program's structure. The primary goal of this thesis is to analyze carefully the performance of the Technology Chess Program as a paradigm for analysis of other complex AI performance programs. The analysis uses empirical, analytical, and statistical methods. The empirical explorations included entering TECH in U.S. Chess Federation tournaments, earning an official USCF rating of 1243 (Class D). This rating is used to predict the performance of faster TECH-like programs. Indirect rating methods are considered. TECH's performance is compared with the moves made in the Spassky/Fischer 1972 World Championship match. TECH's tree-searching efficiency is compared with a theoretical model of the alpha-beta algorithm. The search is found to be much closer to

perfect ordering than to random ordering, so that little improvement can be expected by improving the order of moves considered in the tree. An analysis of variance (ANOVA) is performed to determine the relative importance of the tactical heuristics in limiting a tree search. Sorting captures in the tree is found to be very important; the killer heuristic is found to be ineffective in this environment. The major contribution of the thesis is its paradigm for performance analysis of AI programs. Important individual contributions include the use of ANOVA to assign credit to program mechanisms, direct and indirect methods for determining playing ability, the use of an analytical model to show how close the search is to perfect ordering, and a new interpretation of the Samuel coefficient.

Goodman, R. G. Analysis of languages for man-machine voice communication. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, May, 1977.

Comparing the relative performances of speech understanding systems has always been difficult and subject to speculation. Different tasks naturally require different vocabularies with varying acoustic similarities. Moreover, constraints imposed by the syntax may make recognition easier, even for vocabularies with high ambiguity. This thesis presents an analysis of ambiguity, restriction and complexity in speech understanding system languages. The ambiguity considered involves the similarity of acoustic signals and the ambiguity it causes at other levels of recognition. Phonemes spoken in isolation are misrecognized by both man and machine. Words and phrases having similar phonetic structure are confused. This confusion increases the complexity at the phonetic, lexical and syntactic levels. Ambiguity may also occur at the semantic and user discourse levels. The concepts presented here can be extended to these levels. Measures are developed which permit the relative comparison of the difficulties of a given set of recognition tasks. We present notions of equivalent vocabulary size, branching factor, effective branching factor, search space size and search space reduction. All of these are useful are relative comparison measures. Briefly, the plan of research is to investigate, in order: phonetic ambiguity, word ambiguity, lexical ambiguity, syntactic constraint and the combined effects of lexical ambiuity and syntactic constraint. First, the major source of ambiguity, the acoustic speech signal itself, is considered. Several measures for quantifying phonetic ambiguity are investigated and compared. These measures provide a basis for the computation of lexical and phrasal ambiguity. A

model for lexical ambiguity is presented which utilizes the knowledge of phonetic ambiguity and a general representation of the vocabulary to estimate the probability that an acoustic realization of some sequence of idealized phonemes will result in incorrect recognition. The average expected number of words retrieved in an syntactically unconstrained lexical search is computed from these probabilities. This number is called the equivalent size of the vocabulary. The 10 digits, for instance, have an equivalent size of 1.19 words, while the equivalent size of the spoken alphabet ("a", "b",..."z") is 3.87. The syntax of languages for speech understanding systems imposes re- strictions on the number of word pairs, triples, etc. which can occur in the language. These limitations can dramatically reduce the total size of the search space. One of the languages investigated has a 250 word vocabulary and an average sentence length of 8 words. Syntactic restrictions reduce the branching factor to 7.3. This is, on the average one must disambiguate among 7 words. Equivalent vocabulary size may be viewed as a branching factor in the case where there are no syntactic constraints. Thus, lexical ambiguity and syntactic restriction are measured in the same terms. This unification allows combined effects of vocabulry ambiguity and syntactic complexity to also be viewed as a branching factor. Two models for complexity of connected speech are defined. A "best" behavior model which assumes that word boundaries are known and therfore the only confusions that may arise are when two (or more) phonetically similar words have the same contexts. The effective branching factor obtained can be viewed as an optimistic representation of the expected behavior of the system. A "worst" case model is also discussed. The important contribution of this thesis is that is provides a way to characterize the reltive difficulties and accomplishments of different speech understanding systems. Vocabulary size is not a good measure of lexical complexity; some other measure of vocabulary size, normalized for relative ambiguity would be better. The number of production rules is not a useful measure of grammatical complexity. In fact, quite the opposite may be true; more rules imply more constraint. Some other measure, such as the average number of alternatives at each choice point would be better.

Hayes-Roth, F. and McDermott, J. Knowledge acquisition from structural descriptions. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

The representation of concepts and antecedent-consequent productions is discussed and a method for inducing

knowledge by abstracting such representations from a sequence of training examples is described. The proposed learning method, interference matching, induces abstractions by finding relational properties common to two or more exemplars.

Hayes-Roth, F. and Lesser, V. R. Focus of attention in the Hearsay-II speech understanding system. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, September, 1977.

Hearsay-II is a complex, distributed-logic system for speech understanding developed at Carnegie-Mellon University. Processing is performed by independent, data-directed knowledge source processes that examine and alter values in a global data baxe representing hypothesized phonetic segments, syllables, words, and phrases, as well as the hypothetical temporal and logical relationships among them. The question of how the numerous potential activities of the knowledge sources should be scheduled to complete the interpretation of an utterance in minimal time is called the "focus of attention problem." Near optimal focusing is especially important in a speech understanding system because of the very large solution space that potentially needs to be searched. Thus, this focus of attention problem is representative of general resource allocation problems involving cooperative and competitive processes. Using the concepts of stimulus and response frames of scheduled knowledge source instantiations, competition among alternative responses, goals, and the desirability of a knowledge source instantiation, a general attentional control mechanism is developed. This general focusing mechanism facilitates the experimental evaluation of a variety of specific attentional control policies (such as best-first, bottom-up, and top-down search strategies) and allows the modular addition of specialized heuristics for the speech understanding task. Empirical results demonstrate the effectiveness of the focusing principles, and possible directions for future research are considered.

Kadane, J. P. and Simon, H. A. Optimal strategies for a class of constrained sequential problems. The Annals of Statistics 5 (1977), 237-255.

This paper considers and unifies two sequential problems which have been extensively discussed. A class of sequential problems is proposed that includes both. An arbitrary partial ordering constraint is permitted to restrict possible strategies. An algorithm is proposed for finding the optimal strategy, and we prove that a strategy is optimal for the class of problems if and only if it can

be found by the algorithm. The main tool is a set of functional equations in a strategy space.

Kender, J. An annotated bibliography of natural language and speech understanding systems. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, December, 1977.

This annotated bibliography summarizes some 80 papers dealing with various aspects of natural language and speech understanding systems. Most detail a working system which "understands" "natural" input. Stress is on those issues at or above the level of syntax. Also included are several overviews and criticisms, usually in the form of comparative studies.

Lenat, D. B. and Harris, G. Designing a rule system that searches for scientific discoveries. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, May, 1977.

Some scientific inference tasks (including mass spectrum identification [Dendral], medical diagnosis [Mycin], and math theory development [AM] have been successfully modelled as rule-directed search processes. These rule systems are designed quite differently from "pure production systems". By concentrating upon the design of one program (AM), we shall show how 13 kinds of design deviations arise from (i) the level of sophistication of the task that the system is designed to perform, (ii) the inherent nature of the task, and (iii) the designer's view of the task. The limitations of AM suggest even more radical departures from traditional rule system architecture. All these modifications are then collected into a new, complicated set of constraints on the form of the data structures, the rules, the interpreter, and the distribution of knowledge between rules and data structures. These new policies sacrifice uniformity in the interests of characterization of the task. Rule systems whose architectures conform to the new design principles will be more awkward for many tasks than would "pure" systems. Nevertheless, the new architecture should be significantly more powerful and natural for building rule systems that do scientific discovery tasks.

Lenat, D. B. Automated theory formation in mathematics. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, August, 1977.

A program called "AM" is described which carries on simple mathematics research: defining and studying new concepts under the guidance of a large body of heuristic rules. The 250 heuristics communicate via an agenda mechanism, a global priority queue of small tasks for the program to perform and reasons why each task is plausible (e.g., "Find generalizations of 'primes', because 'primes' turned out to be so useful a concept".) Each concept is an active, structured knowledge moule. One hundred very imcomplete modules are initially supplied,each one corresponding to an elementary set-theoretic concept (e.g., union). This provides a definite but immense space which AM begins to explore. In one hour, AM rediscovers hundreds of common concepts (including singleton sets, natural numbers, arithmetic) and theorems (e.g., unique factorization).

Lenat, D. B. and McDermott, J. Less than general production system architectures. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

Many of the recent expert rule-based systems [Dendral, Mycin, AM, Pecos] have architectures that differ significantly from the simple domain-independent architectures of "pure" production systems. The purpose of this paper is to explore, somewhat more systematically than has been done before, the various ways in which the simplicity constraints can be relaxed, and the benefits of doing so. The most significant benefits arise from three sources: (i) the grain size of a typical rule can be increased until it captures a unit of advice which is meaningful in that system's task domain, (ii) the interpreter can become accessible to the rules and thus become dynamically modifiable, and (iii) meaningful permanent knowledge can be stored in data memories, not just within productions. Although there are costs associated with relaxing the simplicity constraints, for many task domains the benefits outweigh the costs.

Lenat, D. B. The ubiquity of discovery. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, August, 1977.

As scientists interested in studying the phenomenon of "intelligence", we first choose a view of Man, develop atheory of how intelligent behavior is managed, and construct some models which can test out and refine that theory. The view we choose is that Man is a symbolic information processor. The theory is that sophisticatedcognitive tasks can be cast as searches or

explorations, and that each human possesses (and efficiently accesses) a large body of informal rules of thumb (heuristics) which constrain his search. The source of what we colloquially call "intelligence" is seen to be very efficient searching of an a priori immense space. Some computational models which incorporate this theory are described. Among them is AM, a computer program which develops new mathematical concepts and conjectures involving them; AM is guided in this exploration by a collection of 250 more or less general heuristic rules. The operational nature of such models allows experiments to be performed upon them, experiments which help us test and develop hypotheses about intelligence. One interesting result has been the ubiquity of this kind of heuristic guidance; intelligence permeates everyday problem solving and invention, as well as the kind of problem solving and invention that scientists and artists perform.

Lesser, V. R. and Erman, L. D. A retrospective view of the Hearsay-II architecture. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

The Hearsay model has been presented as a paradigm for attacking errorful knowledge-intensive problems requiring multiple, cooperating knowledge sources. The Hearsay-II architecture is the latest attempt to explore the model. This paper describes expriences gained while successfully applying this architecture to the problem of speech understanding. The major conclusions are: 1. The paradigm of viewing problem solving in terms of hypothesize- and-test actions distributed among distinct representations of the problem has been shown to be computationally feasible. 2. A global working memory (the "blackboard"), in which the distinct representations are integrated in a uniform manner, has made it con- venient to construct and integrate the individual sources of knowledge needed for the problem solution. 3. The use of a uniform data-directed structure for controlling knowledge-source activity has made the system easy to understand and modify. 4. A solution has been demonstrated to the problem of focus-of- attention in this type of control environment. This solution does not need to be modified when the sources of knowledge in the system are changed.

McCracken, D. A production system version of the Hearsay-II speech understanding system. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, April,

1978.

A prime candidate organization for large, knowledge-rich systems is that of a production system (PS). PSs are rule-based architectures that have been used successfully for tasks ranging from models of human behavior to large application systems in chemistry and medicine, to classical artificial intelligence programs. The question studied by this thesis is whether a PS architecture (PSA) helps or hinders with respect to implementation problems encountered by Hearsay-II (HSII), a large artificial intelligence system for understanding speech, developed at Carnegie-Mellon University (CMU). This is an important question because many of these problems, such as efficiency, compensating for error, controlling directionality, augmenting knowledge, and analyzing performance, have become limiting factors for performance. To obtain an answer to this question, an actual system (called HSP, for "HearSay-Production system") was implemented on C.mmp, the CMU multi-miniprocessor, with a portion of the HSII speech knowledge translated into productions. An early decision was made to maintain close comparability of HSP with HSII rather than explore the more general question of how to best understand speech witha PS. Two knowledge-source (KS) programs from a complete HSII configuration were completely translated and run in HSP, and these provide a basis for some detailed comparisons between HSII and HSP. Ten other KSs were translated, and their static structure provides supporting evidence. The HSP architecture was heavily influenced by HSII, itself similar to a PSA, and by a general PSA design philosophy manifested at CMU in systems such as PSG, PSNLST and OPS. HSP has several novel features when compared with these three related PSAs, and thus makes a minor contribution in the area of PSA design. The main results of the thesis are presented as a list of 17 assertions organized into five categories: Representation and Architecture, Space Efficiency, Time Efficiency, Parallelism, and the Small Address Problem. The HSP architecture is found to be adequate for representing the HSII speech knowledge,even though HSPis simple compared to other PSAs. Space and time efficiency are another matter. There is a moderate space penalty for representing declarative HSII knowledge as HSP productions, which is cause for concern since HSII contains many large declarative knowledge structures. Evenmore serious is the

substantial space inefficiency of the global HSP working memory, since it must be used in place of large, highly optimized local working memories typically used by HSII KSs. HSP's lack of local working memory results also in a large loss of time efficiency because of heavier useof data-directed control and greater creation/read/write costs

in its global Working Memory. In the two-KS configuration this loss is a factor somewhere in the range 6 to36, but projecting to a full KS configuration yields a much larger factorof 100 to3000 since many of the KSs to be added make heavy use of local working memory and control(in their HSII form). Some of the time efficiency handicap is made up through increased parallelism of HSP over HSII. A source of parallelism not exploited by HSII, called intra-KS parallelism, results from HSP's smaller knowledge unit size. We estimate conservatively a half order of magnitude increase in parallelism for a full KS configuration. It could be much greater than that if HSP's less powerful synchronization mechanisms turn out tobe adequate with a full complement of KSs. Finally, HSP is found to aid solution of the Small Address Problem, as it exists on C.mmp, by making it easy to do overlaying of both long-term knowledge and working memory. The thesis concludes with brief discussion of 9 important questions which have emerged from the current study -- questions which must be answered to complete the evaluation of a PSA for HSII.

McDermott, J., Newell, A. and Moore, J. The efficiency of certain production system implementations. In Pattern-Directed Inference Systems. (Hayes-Roth, F. and Waterman, D., Ed.) Academic Press, New York, NY, 1978. Also CMU-CSD Technical Report September, 1976.

The obvious method of determining which productions are satisfied on a given cycle involves matching productions, one at a time, against the contents of working memory. The cost of this processing is essentially linear in the product of the number of productions in production memory and the number of assertions in working memory. By augmenting a production system architecture with a mechanism that enables knowledge of similarities among productions to be precomputed and then exploited during a run, it is possible to eliminate the dependency on the size of production memory. If in addition, the architecture is augmented with a mechanism that enables knowledge of the degree to which each production is currently satisfied to be maintained across cycles, then the dependency on the size of working memory can be eliminated as well. After a particular production system architecture, PSG, is described, two sets of mechanisms that increase its efficiency are presented. To determine their effectiveness, two augmented versions of PSG are compared exprimentally with each other and with the original version.

Perdue, C. and Berliner, H. EG -- A case study in problem solving with king and pawn endings. Proceedings of the Fifth

International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

We present an overview of the design of a program that plays simple chess endings with pawns and details of interesting aspects. The program evaluates positions according to production-like rules and also generates moves through the mediation of rules that produce "strategies". Effects of the design are discussed, partly through examples. The design affects the application of standard chess programming principles, among them use of cutoffs, the definition of a repeated position, and the comparison of values of positions. We also describe problems and solutions of problems concerning concepts peculiar to this type of design, especially the concept of search within the context of pursuing a particular strategy.

Reddy, D. R. and Rubin, S. Representation of three-dimensional objects. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, April, 1978.

This paper describes a system for generation and representation of computer models of arbitrary three dimensional objects. Three representations are explored which have varying time and space tradeoffs. These representations, which are tailored for irregular 3-D objects, store every input point exactly by using a tree structuring of the object space. Their advantage over more common representations is that they make perspective and hidden-surface elimination less difficult. In addition, one of the representations simplifies the storage of dynamic objects and objects with redundant sub-parts.

Robertson, G., Newell, A. and Ramakrishna, K. ZOG: A man-machine communication philosophy. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, August, 1977.

ZOG is a rapid response, large network, menu selection system used for man-machine communication. The philosophy behind this system was first developed by the PROMIS (Problem Oriented Medical Information System) Laboratory of the University of Vermont. ZOG will be used in a number of task domains to help explore and evaluate the limits and potential benefits of the communication philosophy. This paper discusses the basic ideas in ZOG and describes the architecture of a system that has just been implemented to carry out that exploration and evaluation.

Rychener, M. Production systems as a programming language for artificial intelligence applications (3 volumes). Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, December, 1976.

This thesis develops a system architecture for artificial intelligence (AI), called production systems (PSs). Each production is a simple condition-action rule, with conditions stated on a global Working Memory and actions consisting primarily of simple modifications to that memory. Actions can also consist of forming new productions. PSs have been applied to a limited extent in computer science and to a somewhat larger extent to specialized studies in AI. They are used in cognitive psychology to model human intellectual capabilities at a detailed level. With AI research tending toward larger systems with greater flexibility requirements, PSs are promising as candidates for the primary knowledge encoding medium, but certain questions and problems with PSs have been raised. The questions revolve around the practical feasibility of PSs for building large systems in a diversity of task domains, the preservation of desirable PS properties when they are applied to much larger systems than previously, and the specific advantages and disadvantages of PS architectural features. This thesis seeks answers to such questions by constructing PSs to perform the following tasks, all of which have been developed by past AI research: extracting equations from typical high school algebra story problems (Bobrow's STUDENT); learning lists of nonsense syllable pairs (Feigenbaum's EPAM); solving a variety of puzzle tasks using a single set of general methods and processes (Newell, Shaw, Simon and Ernst's GPS); playing a simple class of chess endgames (Perdue and Berliner); discoursing in natural language about a toy blocks scene (Moran's mini-linguistic system); and solving toy blocks manipulation problems (Winograd's SHRDLU system). Each implementation is analyzed to bring out PS characteristics. Evaluations of PSs as a programming language are made according to the traits: practical feasibility, style, degree of theory-boundness, power and overhead of expression, productivity, efficiency, architectural flexibility, and level. A taxonomy of control is presented, and measures of frequencies of usages in the PSs of various forms of control in that taxonomy are used to support the discussion of power and overhead of expression. The actual PSs are able to effectively exploit PS power in the particular areas of selections and iterations. Specific features of

the particular language design used here are central to the capabilities discussed. A taxonomy of representation is

developed, to provide a basis for adding openness to the PSs, replacing ad hoc internal naming conventions, and to allow measurement of the modularity of PSs, making interdependencies of various parts more examinable. The taxonomy of representation is applied to one of the larger PS programs with the finding that the split between inter-module assumptions and intra-module assumptions is roughly an order of magnitude, approximately the form of a nearly decomposable system. PSs are found to be effective and advantageous for the programming constructs typical of AI systems. They have particular advantages in style, conciseness, and architectural flexibility. Major successes can be expected in applying PSs to large-scale understanding systems of the sort currently being explored. They are particularly useful in domains where system knowledge must grow dynamically through interaction with humans and with a task environment, but without the expense of analysis of how each new piece of knowledge must fit into existing structure. Their diversity of application and their problem-solving capabilities, both of which are deemed essential to building understanding systems, have been adequately demonstrated by this thesis.

Simon, H. A. Identifying basic abilities underlying intelligent performance of complex tasks. In The Nature of Intelligence. (Resnick, L. B., Ed.) Lawrence Erlbaum Associates, Hillsdale, NJ, 1976, pp. 65-98.

This chapter is listed under the heading of "computer simulation approaches to the study of intelligence." Another section of the book is labelled "processes in intelligence." This division of labor is only pragmatic, for computer simulation of intelligence is simply a particular technique for identifying and studying processes in intelligence. In fact, this chapter is not restricted to simulation techniques either, because these techniques are used most profitably in close conjunction with exprimentation and with observation of human behavior. A simulation is just one phase in a cycle of observing and exprimenting, building and testing theories. The simulation is both a formal expression of the theory and a means of inferring consequences that can be tested empirically.

Simon, H. A. and Kadane, J. B. Problems of computational complexity in artificial intelligence. In Algorithms and Complexity: New Directions and Recent Results. (Traub, J. F., Ed.) Academic Press, New York, NY, 1976, pp. 281-300.

Our purpose in this paper is to consider the relevance of questions of computational complexity, or questions

analogous to these, to the theory of search in problem-solving domains. To this end, we will survey what is known about measuring the sizes of problem spces and the power and efficiency of algorithms. Since our aim is to characterize this area of investigation and to interpret results rather than present them in detail, we will state the results informally and will omit proofs. Typical problems in computational complexity may be characterized as follows: We have a one-parameter domain of computations, where the parameter, N, say, is a measure of computation size (e.g., the order of a matrix to be inverted). We have also a basic set of operations (e.g., addition, multiplication). We wish to determine the minimum number of operations required to carry out computations from the sub- domain of size N, for all N. This minimum number expresses the complexity of computations belonging to the domain as a function of N. Notice that the measure, as stated, is unambiguous only if all computations of size N in the domain require the same number of operations. If that is not the case, we might change the problem slightly, e.g.: For each N, determine the maximum number of operations required to carry out any computation in the subdomain of size N, in each case using the algorithm that performs that computation in a minimum number of operations. The approach to complexity problems may be constructive. We may seek to find specific algorithms that solve problems of a given domain with a minimum number of operations, and we may seek to compare the relative efficiencies of different algorithms, whether or not any of them is known to be maximally efficient. In the latter case we face the difficulty that measures of relative efficiency are seldom invariant over all poblems of a specified class. Then some kind of expected value based upon a probability measure defined over the problem domain is needed.

Simon, H. A. On the nature of understanding. In Perspectives on Computer Science. (Jones, A. K., Ed.) Academic Press, New York, NY, 1977, pp. 199-216.

The term "understanding" is one of those words -- along with "intelligence", "problem solving", "meaning", and the like -- that we have borrowed from the vocabulary of everyday life, and that we apply in the field of artificial intelligence without assigning technical definitions to them. There is no harm in this provided that we do not imagine that because a word "understanding" exists, there must also exist some unitary aspect of human intellectual behavior to which the term refers. Indeed, an artificial intelligence, the very vagueness of the notion of understanding has been of some use. Investigators have

tried to penetrate its meaning by constructing programs that would behave like a person who understands in some class of situations. Different investigators have chosen different areas of behavior in which to test their systems; the last decade has seen a substantial number of interesting varieties of intelligent behavior, with a consequent enlargement of our grasp of what information processes and what organizations of those processes are required for understanding. As is typical of this kind of artificial intelligence research, at first we gradually accumulate empirical examples -- running programs -- that exhibit the phenomena of interest; then we analyze these programs for common mechanisms and common structures; and finally, we try to construct generalizations about what kinds of mechanisms are essential to produce the phenomena, and how these mechanisms produce their effects.

Simon, H. A. Artificial intelligence systems that understand. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August, 1977.

From its beginnings, artificial intelligence has borrowed freely from the vocabulary of psychology. The use of the word "intelligence" to label our area of research is a case in point. Other terms referring originally to human mental processes that have considerable currency in AI are "thinking," "comprehending," and, with increasing frequency in the past five years, "understanding." In fact, these terms are probably used more freely in AI than in experimental psychology, where a deep suspicion of "mentalistic" terminology still lingers as a heritage of behaviorism. It is not my intent to engage in a barren lexicographic exercise, nor to bait those among us who are aroused to indignant emotion whenever terms from human psychology are used in reference to computers. We employ these anthropomorphic terms because we find them useful in defining our research goals, and therefore it is important that we attach clear operational meanings to them. Before discussing computer programs that understand, we need to consider how we can judge whether such programs are successful.

Stickel, M. E. Mechanical theorem proving and artificial intelligence languages. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, December, 1977.

This dissertaton is principally concerned with

incompleteness issues in the design of artificial intelligence languages. Major sources of incompleteness are the pattern matching and inference facilities of the languages. Incompleteness in the area of pattern matching can be repaired by developing unification algorithms for the specialized data types of the languages. A complete, but potentially infinite unification process is described for arbitrary data types in general and is applied to the QA4/QLISP vector, bag, and class data types. Finite, complete unification algorithms are also described for the bag and class data types. The bag unification algorithm is extended to the case of unification of first order predicate calculus terms with functions which are both associative and commutative. Imcompleteness in the area of the inference system can be repaired by use of some form of the p inference procedure which is a complete extension derived from model elimination of the problem reduction method. This can readily be accomplished in present or new artificial intelligence languages by attempting to derive all goals in the context of the asserted negations of all higher goals. The problem of compatibility of the p procedure and use of models is addressed though not finally resolved. Design of and experimental results for the theorem proving program PSTP are presented. The inference system used is a variant of the p procedure and a version of PSTP was used to verify the effectiveness of the special unification algorithms developed.

5. SOFTWARE TECHNOLOGY (SFT)

Software Technology (SFT): The major problem of concern is: How to design the basic software tools and methodology with which to create computer systems, both hardware and software? The current foci are on using symbolic descriptions of computer systems to automate and generalize central activities in computer science and engineering (SMCD: Symbolic Manipulation of Computer Descriptions), on exploring important properties of operating systems, and on the development of software implementation systems. Research in the SMCD area focusses on the development of two important specific applications: a production quality compiler-compiler and a computer aided design system for design from Register-Transfer level module sets.

5.1 Introduction

We conducted a broad investigation in how to obtain a software technology that is capable of producing high quality systems efficiently and predictably, with a strong focus on the symbolic manipulation of computer descriptions.

Specifically, in working with symbolic descriptions of computers we: (1) are developing a production quality compiler-compiler; (2) developed a computer aided design system for RT-level modules; and (3) used the ISP computer description languages to describe and evaluate quantitatively architectures in a cooperative effort with NRL and ECOM.

Specifically, also, with respect to software systems we: (1) developed and tested the important contributions of Hydra, the operating system for the C.mmp multiprocessor system, to multiple suboperating systems and flexible protection structures; (2) studied the implementation of Alphard, a new programming system that permits free use of abstraction, and tested its ability to support verification; (3) developed further the interactive symbolic implementation system, L*, and demonstrated its effectiveness; and (4) demonstrated the Algol 68 generalized multiprocessing primitives and the machine independence of its runtime environment.

5.2 Background: Pre-Contract Period

Software technology (SFT) is concerned with all aspects of the construction of hardware/software systems. Specifically, it aims at obtaining "high quality" systems: which are produced on time, which are produced to budget, which are correct, which are efficient, which can be maintained and enhanced, and which are tolerant of their human users. Software technology is composed of both the principles and knowledge (the know-how) to be used in producing software, and the tools (primarily software systems themselves) used in the production of software -- compilers, debuggers, editors, design systems, etc.

The practical implications of software technology are immense. For example, roughly 70% of the current DOD data processing budget is devoted to software, 30% to hardware. By 1985 it is projected that the software component will exceed 90%. Since the cost of hardware is dropping rapidly these percentages need not be disturbing in themselves; however, coupled with the fact that much of the present software is late, over-budget, incorrect, virtually unmaintainable, inflexible, and so on, it IS a cause for concern.

By its nature, software technology is both applicable to, and overlaps the interests of most other subfields of computer science. Of particular interest, however, are:

- Programming Methodology: the study and development of design and construction methodologies, such as "structured programming", which attempt to match the complexity of programs to our human (in)ability to cope with that complexity.

- Programming Tools: the study and development of programming tools, such as programming languages, operating systems, utilities, etc., which simplify the program development task.

- Programming Concepts: the development of concepts, such as procedures and coroutines, in terms of which specific programs may be more easily and clearly expressed.

- Program Testing and Verification: the development of formal and informal methods for determining whether a given program performs as specified.

- Algorithm Development and Evaluation: the development and evaluation of new algorithms for the solution of specific problems.

At present the construction of large programs is at best an art. We cannot reliably predict the cost, completion date, size, or speed. We cannot even be certain that the program is feasible, and observing failure we cannot always determine whether a different approach might have succeeded. Nevertheless, progress has been made: large programs are routinely built, and tools and methodologies have been developed and used with success.

Although the name "software technology" has been coined fairly recently, many of the activities listed above have been central to computer science from its inception. For example, programming tools, especially operating systems and programming languages, have been studied for over twenty years. There are at least two reasons for the persistence of these topics: first, the programs involved are themselves interesting objects of study, and second, operating systems and languages, in particular, are vehicles which carry the most recent concepts of the science.

Consider programming languages, for example. Compilers, the programs which translate a program written in some programming language into a form which may be executed, are large, complex, and used by a large community. Every computer manufacturer is obligated to construct several for each computer design sold. Thus, the construction of compilers represents an application in which one, or all, of the aspects of software technology can be exercised. Languages themselves are the notational devices used to communicate programs between people and between people and the machine; as our understanding of what is to be communicated, and of better ways to communicate it, has expanded, languages have evolved to incorporate this better understanding.

The central difficulty faced by software technology can be summed up in a single word: "complexity". Because the unit operations performed by a computer are exceedingly primitive, even relatively simple external behavior can be achieved only by complex synthesis of these primitive actions. We have come to realize that this synthesis is usually more complex than the human mind is able to deal with in detail. Hence the specific topics listed above

are, in one way or another, all concerned with reducing the "apparent complexity" of programs, ie, the complexity perceived by the programmer.

The reduction of "apparent complexity" generally comes in one of two flavors: providing explicit abstractions which capture complex notions, and providing mechanisms which permit the definition of new, task-specific, abstractions. Programming methodology, for example, is concerned with techniques for permitting isolation of concerns (so that the programmer may focus on only one aspect of the program at a time and yet be assured that the whole will function properly). This methodology is essentially concerned with the identification and definition of task-specific abstractions. Programming tools are all concerned with raising the level at which the programmer works, and delegating irrelevant detail to the tool. The development of programming concepts focuses on finding clearer, more concise ways of expressing a class of programs. Thus, they are concerned with the identification and definition of broadly applicable abstractions.

In many cases we have had neither the appropriate abstraction nor the appropriate mechanism for defining it. Only recently, for example, has the need for the symbolic description of computer hardware been recognized for other than simple simulation. Thus, neither the appropriate abstractions nor the means for defining them has existed (the classical abstractions consisted of circuit diagrams and programming manuals, neither of which permits, for example, the automatic design of either compilers or hardware realizations.)

The long range goal of software technology, and the CMU program in particular, is to increase our ability to predictably produce high quality hardware/software systems. Progress toward that goal may be measured either in terms of an increase in the quality of particular systems (eg, compilers), or an increase in the complexity of tasks we may produce at a given quality.

Because of the nature of software technology it is impossible in a research effort to cover all the aspects listed above or yet to exclude any of them in a given task. Thus we choose to focus on a set of tasks, each of substantial

intrinsic interest, but each also containing a component of all of these aspects.

The CMU activities within software technology fall under two broad categories with, nevertheless, strong coupling between them: symbolic manipulation of computer descriptions (SMCD), and software systems (SS). The SMCD effort includes a number of activities related to the description of computer systems, and the manipulation of these descriptions in various ways (for example, their use in automatic realization in hardware and in the automatic generation of compilers for that machine). The SS effort includes a number of activities aimed at obtaining a deeper understanding of appropriate abstractions (for example the appropriate means of stating and understanding parallel algorithms) and abstraction definitions.

Each of the specific tasks undertaken are discussed in detail in the section on "Accomplishments 1976-78"; here we shall state the scientific goals of these tasks. Although there is nearly a one-to-one relation between goals and tasks, it should be remembered that each task is viewed locally both in terms of its primary goal and as an instance of a software system to which the results of software technology should be applied.

The primary scientific goals are:

- To produce, within five years, a "production quality compiler-compiler"; a program which, given a language and computer description, produces a compiler competitive with current hand-crafted compilers. The major scientific achievement will be the invention of algorithms (and heuristics) for converting the symbolic descriptions into a compiler.

- To produce the technology, realized as a program, to convert a symbolic computer description into an efficient hardware realization. This implies the invention of algorithms, but also the determination of appropriate symbolic (language) and hardware (modules) abstractions.

- To further enhance and evaluate the tools used in software implementation, and specifically to incorporate the results of other aspects of software technology into these tools and to test the hypotheses on which they are predicated.

5.3 Relevance of Research

It has become increasingly clear that software costs will dominate hardware costs in the future. It has already been mentioned, for example, that software is expected to consume 90% of the DOD data processing budget by 1985. It has also become generally recognized that the current result of this expenditure is too often software that is late, is bigger and slower than predicted, contains far too many errors, and so on. The objective of software technology is to permit the predictable construction of large hardware/software systems on time, within the budget, without residual errors, and meeting performance specifications.

This objective will not be met by the discovery of some single organizing principle. Software construction is fundamentally an engineering activity on a large, complex system. All aspects of the construction activity (and of the object system) must be brought under control in order that the objectives above be realized in practice. Many diverse scientific results and engineering principles are involved.

The SFT program at CMU focusses on a particular set of the possible fronts which might be pushed, a set we believe has the potential for high payoff. In discussing each of these below we first focus on the payoff, then try to show how the particular proposed (and ongoing) effort leads to this.

Production Quality Complier Complier : The availability of a PQCC technology will have several important effects. First, it will reduce the cost of compiler construction (from 2-10 man years to a few man-months). Second, it will eliminate the residual errors found in all extant compilers. Third, it will reduce the elapsed time to produce a compiler from years to months. Fourth, it will make the use of higher-order languages feasible in contexts where currently assembly language is used (either for efficiency or to avoid the additional time and cost of constructing a compiler).

The particular strategy we chose to develop the PQCC technology involved constructing a series of compilers. Each is a useful (relevant) product in

its own right, and each involves greater use of automated production tools than its predecessor. The first step was to produce a new, optimizing Bliss compiler for the PDP-10. This is a compiler already in commercial use, and this step would provide a significant upgrade of the existing system. The second step was to produce a Bliss compiler for the IBM 370 but has been changed to the PDP-11. This will demonstrate clearly the cost (in real time and in effort) of producing a significant compiler on a new machine. The final step was be to produce a compiler for a different language (Pascal) for the PDP10. This would demonstrate the generality of the technology and will finally open up the possibility to applications throughout DoD.

Computer Aided Design Systems for RT-level Modules : This research is essentially the hardware analog of the PQCC effort and has similar payoffs. The rapid evolution of hardware (device) technology makes it mandatory to achieve architectural definitions which are independent of a specific technology; this, in fact, is the premise of the CFA project (see below). Success in this endeavor will permit the military to define an architecture which persists through several generations of dominant device technologies, and to rapidly and reliably create implementations of that architecture from the most appropriate technology for a given application. Thus, for example, the expensive and time consuming tasks of reprogramming both applications and support tools (eg, assemblers, loaders, etc.) can be avoided.

The relevance of our CAD effort rests on four aspects. (1) It takes as input the ISP, ie, the instruction set architecture. Thus it works with the level that one wants to keep invariant for practical reasons (as noted just above). (2) It designs in terms of commercially available module sets, not components which are no longer of practical interest for real design. (3) It designs large enough computer systems to be of practical interest, namely PDP11-sized systems. (4) It incorporates procedures that attempt to optimize the structure of the design, not just to produce feasible designs, thus it attempts to produce designs that are good enough to be practical. Reaching a

system with this combination of characteristics is difficult, but we deliberately set our goals high enough to have practical implications.

ISP description and evaluation of architectures : The ultimate goal of the SMCD effort, of which this is a piece, is to provide symbolic descriptions of computers which may be manipulated in task-specific ways. One such task is the evaluation of proposed architectural designs. Given a precise definition of a set of machines it will be possible to (1) make quantitative comparisons between them, (2) determine whether a particular implementation is faithful to the definition, and (3) analyze and localize bottlenecks of particular architectures (with a view to making local changes to them).

The immediate impact of this effort on DOD is evident in our cooperation with NRL and ECOM on the CFA project (which is attempting to select a common computer architecture for most military applications). Success in this project will avoid duplication of both systems and applications software.

The CMU ISP compiler and simulator was used to gather statistics on benchmark programs written for the four semifinal architectural candidates. These statistics were used to compare the candidates and select a finalist. The ISP definition of the finalist has become part of the procurement document (thereby avoiding misinterpretations of the machine specifications by vendors). Eventually, ISP specification may become part of a general computer procurement policy thus insuring a methodology for determining whether vendors have met contractual specifications. The ISP compiler and simulator will also be used to test proposed modifications to the standard architecture and measure their software impact.

Alphard as a Computer Description Language : The basic arguments for the relevance of Alphard as a computer description language are the same as those for ISP above. We continue to push the development of ISP because of its maturity. However, there are a class of highly relevant tasks which it cannot handle; Alphard is our response to these tasks. These tasks include: (1) design verification, (2) multi-level modeling, and (3) integrated

hardware/software design.

Current computer description languages are procedural and operate with a fixed set of primitive semantic notions. Consequently the problem of verification of a large design is impractical for essentially the same reasons as verification of large software designs is impractical -- in fact the parallelism in machine descriptions exacerbates the problem. The fixed semantic basis prevents detailed design below the RT level and, because of the size of the definition, also prevents ascending into the traditional software domain.

If Alphard meets its goals with respect to computer description (note that a test of this is part of the proposed effort), it will be possible to provide a priori verification of architectural designs. Further, the decomposition methodology on which the language is based provides non-procedural specifications of components; this will permit far greater flexibility in implementation and, in particular, allows arbitrary division into hardware, firmware, and software. Finally, the semantic basis of the language is such that design below the RT level is practical when the situation warrants.

Hydra : Many aspects of the Hydra development are highly relevant to DOD; we shall mention three: (1) protection, (2) policy/mechanism separation, and (3) methodology demonstration.

The relevance of protection and security to the military (and the rest of society) is hardly an issue. There are two aspects of the Hydra contribution which deserve special emphasis: flexibility and coverage. The Hydra mechanism permits the dynamic definition of new classes of protected objects and the security policies which govern their use. These policies need not be hierarchically ordered (as in confidential, secret,...). Thus policies such as "need to know" are naturally expressed. User identification (passwords and the like) is not a system function; thus arbitrary protocols may be established to insure authorized access to specific units of information. Moreover, many traditional "protection problems" which cannot be solved in

current authority-based protection problems have been shown to be solved by the Hydra mechanism.

Hydra is at the "demonstration" phase of a concept demonstration; the reported effort includes several tests of these capabilities.

The policy/mechanism separation in Hydra permits most resource allocation policies to be safely defined by user-level, non-privileged programs. This facility is relevant in two ways: (1) it permits experimentation with these policies in a multiprocessor environment, thus providing fodder for future designs, and (2) it permits tuning of the systems response to critical applications. We believe this latter is critically important for complex computer systems. A uniform operating system lays "overhead" costs on some of its users precisely because it must treat all users alike in many ways (more precisely, the degrees of adaptation are limited to pre-parametrized dimensions). Such overheads can be costly.

The construction of Hydra was treated as an experiment in software technology. Specifically, we used (1) a higher-order programming language for operating systems, and (2) a particular structured programming methodology. Neither of these are new, but construction of Hydra provides data on the utility of these techniques. Of particular concern are productivity rates (the effort to produce a given sized system) and, more important, the deterioration of productivity as the system grows (ie, it becomes harder and harder to add to an already large system). The results to date are very encouraging. Now that Hydra is reasonably stable, we have moved into the domain of maintenance and enhancement. Each change made at this point provides further data on these techniques during this costly phase of the life of a large system.

Alphard as a Language: As Dijkstra has said, testing can show the presence of errors, but never their absence. Thus we assert the relevance of program verification, that is, of a mathematical proof that a program performs as specified. Applications in the military, eg, secure systems and man-rated

systems, are prime examples where the need for confidence in the correctness of software is paramount.

Alphard combines a programming language design and a proof methodology in the classic divide-and-conquer paradigm. Current program verification technology is capable of dealing with only small programs, and it seems unlikely that a breakthrough will be made which extends their capabilities by several orders of magnitude. Thus, Alphard imposes a language structure which: (1) permits isolated verification of small portions of a larger system, and (2) guarantees that the combination of these portions preserves the conditions for their correctness. The result is that large systems can be verified with existing technology. Again, as with a number of other efforts (PQCC, RT-CAD, Bliss, Hydra), we have deliberately selected our objectives to involve systems of a size to have practical implications if we succeed.

L* system implementation system: Two of the primary gains that L* claims to provide are extremely high productivity and the ability to create total software environments in a hurry, expecially on new (ie, bare) computer systems. Both of these, if striking enough, are extremely valuable for the construction of real systems. As with the work in Hydra, our aim is to demonstrate the size of these gains by quantitative measures on systems of applied interest.

Algol 68: Two aspects of the Algol 68 effort are directly relevant to the DOD: (1) the portable run-time system, and (2) the multi-processing language primitives.

PQCC will permit us to reduce substantially the cost of compiler construction. However, it does not directly address the issue of the run-time system, ie, the collection of software which supports the execution of programs and includes i/o routines, storage allocation, etc. In the current art run-time systems are idiosyncratic to each programming language (and eventto each dialect of a language) and machine. The run-time system of the Algol 68 effort appears to be machine independent (and also essentially

independent of Algol 68). If this proves out (and we will test it), it wil have implications for the portability of software systems.

Another feature of the Algol 68 language consists of a set of primitives for dealing with multiprocessing. These, again, are independent of Algol 68. If they are as successful as we think they will be, these primitives will be useful in adapting algebraic languages generally to multiprocessing environments (eg, even for Fortran-like languages).

5.4 Accomplishments: Pre-Contract Period

CMU has a long and notable history in what is now known as "software technology". The production of software has always been of intrinsic interest at CMU, and we are one of the few universities which has shown a continuing, scientific interest in the practical aspects of software construction. In addition, there has been an operative ethos which dictates that good ideas manifest themselves as programs; thus even theoretical ideas usually appear as runnable software, and the problems of software construction are a part of every researcher.

The history of accomplishments at CMU predates both the department and the ARPA contract, and includes the development of some of the first algebraic programming languages (IT,GATE), the development of the first list-processing languages (IPL2-5), participation in the definition of Algol 60, and early involvement in automated parsers (Floyd-Evans productions). CMU was also involved at an early stage in remote-batch systems (on the G-21), general parsing algorithms (Earley's algorithms), conversational programming languages (LCC), program verification (Floyd's assertion method and King's program), extensible languages (PPL), systems implementation languages (Bliss and L*), programming methodology (Parnas' modular decomposition and the elimination of the goto from Bliss), compiler optimization (Bliss), and so on.

A complete list of CMU accomplishments in the software technology area is beyond the scope of this report, hence we shall focus on those relatively recent accomplishments which bear most directly on the reported effort.

Bliss (language): Bliss was one of the first programming languages designed specifically for use in the development of systems software, and along with BCPL remains one of the most widely known and used. It pioneered the user-definition of the representation of data structures and the elimination of the "goto" as an explicit control mechanism. It has been adopted by Digital Equipment Corp. (DEC) for its software development, by Bell Labs for ESS support, and is also in use at roughly 30-50 other university and commercial sites for production software development.

Bliss/11 (compiler): Bliss/11 is the first (and only) highly optimizing compiler developed in a university, and the quality of its code is a major reason for the acceptance of Bliss by practical systems developers. Direct comparison of the quality of the object code produced by compilers for diverse languages and machines is difficult, but it appears that the Bliss/11 compiler is substantially better than any other currently available. The reasons for this superiority are largely related to the organization of the compiler, which represents a major contribution and is detailed in a monograph ("The Design of an Optimizing Compiler", Wulf, 1975). Further work on specific aspects of this decomposition appear in theses by Geschke (on global flow analysis), Newcomer (on generating optimal code sequences from a machine description), and Johnsson (on machine-independent optimal register allocation); all of these bear directly on the PQCC effort.

ISP/PMS: ISP and PMS are languages originally designed by Bell and Newell to describe computer systems, and were used in their book on computer structures. ISP was designed to describe the instruction-set of the computer; PMS was designed to describe the interconnection of major components (processors, memories, i/o devices, and the switches between them). ISP has been adopted by DEC to define its processors precisely, and by the armed services to define a common processor for future military applications (the CFA, or "computer family architecture", project). ISP has been used by Barbacci as input to a program which synthesizes a hardware realization of the

processor described by the ISP; at the start of the 76-78 contract period this program was limited in both the size of the processor which can be described (to roughly a PDP-8 sized machine) and the quality of the realization. Both were expanded in the reported effort.

Hydra: Hydra is the operating system designed to run on C.mmp (described elsewhere). It represents a number of significant developments in the operating system field, specifically:

- A (more) flexible protection structure, i.e. one which is extensible, and in which a user may define new type of objects to be protected and the nature of the protection applicable to each.
- A "good" protection structure in the sense that a wide range of security policies may be defined.
- The ability to permit user-level programs to define resource allocation policies, e.g. scheduling, disk management, etc.
- The "right structure" hypothesis -- the notion that the specific structure of the facilities provided by Hydra permits enhancement in an easy manner.

Although a few features remain unimplemented, and some redesign of various pieces is desirable, Hydra is essentially complete. It operates effectively with the present configuration of C.mmp (16 processors). Overheads appear comparable to, or lower than, uniprocessor operating systems. New object types, and their associated protections, have been defined. Resource allocation is indeed controlled by a user-level program. Present data indicates that: (1) programmer productivity has been 2-4 times greater than on comparable research systems, and (2) productivity has been linear with program size (indicating that the complexity of modifying and enhancing the system has not increased as the system matured).

We considered, however, that the goals (hypotheses, really) listed above had not been subjected to thorough testing, and the reported effort included further tests of them.

L*: This is a software implementation system which is quite different from the main-line, higher order language systems (eg, Bliss and BCPL). It is

completely interactive; it is a symbolic manipulation system; it grows its object system within itself, rather than producing object modules; it provides a total environment for programming (coding, editing, compiling, debugging, monitoring ...). It many respects it attempts to apply the lessons learned in building MI systems (with list processing languages, eg, Lisp) to building systems generally. L* has been operational in various versions since 1971, being used by a very small experimental community.

Although existing programming systems leave much to be desired, the introduction of a new one still requires substantial justification, preferably by quantitative data on how well it performs. We have begun a number of experiments on productivity (instructions/man-day) and portability (effort to move the system to a different machine). The productivity results appear to be very good (many times industry norms). We have one data point on porting L* to a new stand-alone system (C.mmp) and to a stand-alone microprocessor at Xerox PARC.

Algol 68: A compiler for Algol 68 has been built for C.mmp/Hydra. This project has not been mentioned explicitly in prior proposals since it has been treated as a low-priority, lean-manpower project. We mention it now because it represents both a demonstration of the effectiveness of the software technology effort at CMU, and because this implementation will serve as a vehicle for multiprocessor programming studies during the coming two years. Algol 68 is a large (some say baroque), language incorporating most of the newest concepts in programming languages; its implementation has posed substantial problems, and several universities have devoted 5 years to it without success. Our implementation (which is a slight subset of the full language) was written in Bliss, will eventually run on both the PDP-10 and PDP-11 (only the 11 version is complete), and involved about 2 man-years. It is currently available on C.mmp, and includes multiprocessing support.

5.5 Accomplishments: Contract Period 1976-78

5.5.1 PQCC: production quality compiler-compiler

The goal of PQCC is the production of a compiler-compiler that produces high quality code, and produces compilers for arbitrary target machines by working from symbolic descriptions. By SEPT78, complete specifications of the compiler modules were produced using as its basis the Bliss/10 compiler.

In conjunction, a major breakthrough in the automation of code generation was made by Cattell (1978). By developing a high level computer description language, and the application of MI heuristic search methods, Cattell was able to construct optimizing code generation templates for program trees, hence, demonstrating the feasability of production automatic code generators.

Work is continuing on the other aspects of the problem with the goal of automatically producing a Pascal compiler for the PDP10, and a Bliss compiler for the PDP11.

5.5.2 Computer aided design system for RT-level module sets

The design system was extended in three ways. (1) The size of system that can be designed was increased from the present PDP8-sized system to that of a PDP11. This brought us into the realm of large systems from an applications point of view. The system has been tested on the design of a PDP8, producing 70% more states than optimal (2) The module sets, in terms of which the system designs, was extended from the present two (PDP16s and Macromodules) to that of any commercially available Register-transfer level modules. Ie, the Design System will take as input a general symbolic description of the module sets. Currently, the components of the design system are working (funding provided by NSF). This brings us into the realm of real module sets as well as to a point where we can track development of module sets. (3) A simulator was built for ISPS; the new extension of ISP, the computer description language used to specify the target design. This permits evaluating the semantics of the designed system prior to design.

5.5.3 ISP description and evaluation of architectures

The CFA project (Barbacci & Siewiorek, 1977; Barbacci & Parker, 1978), which was a cooperative effort with NRL and ECOM, resulted in a quantitative comparative analysis of existing computer architectures. The comparison demonstrated the adequacy of symbolic computer description (ISP) of real architectures for analysis and planning. The successful completion of the project has resulted in a national standard, allowing for the systematic analysis of production systems.

5.5.4 Alphard: new computer description language

The goal of using Alphard as the next generation computer description language was postponed until the time the Alphard compiler becomes available.

5.5.5 Hydra

Hydra will continue to be improved as the basic operating system for C.mmp. This goes on continuously and shows indirectly in the success of the entire C.mmp enterprise. We cover here specific attempts to extract the important scientific contributions of Hydra and to formulate tests and demonstrations of them.

User-defined policy modules permit the C.mmp to operate in effect as if it had a number of independent operating systems. This offers advantages for the construction of user subsystems and applications, since each application may restructure its operating environment to its own desires. Since all occupy the same physical structure and share its resources, there is a point below which such adaptation is not possible. Of necessity, the gains and limitations of user-defined policy modules can only be studied with genuine user applications, since users are required to invent the new policy modules and to provide the situation in which evaluation is possible. Multiple policy modules have been created for C.mmp. Careful study of these has been made to measure their contributions and costs, both in terms of C.mmp operation and in terms of the users who designed, maintained and adapted them. Also the effects of lack of uniformity in the facilities available to the total C.mmp

user community was studied. The results of these studies is contained in (Wulf & Harbison, 1978).

The protection structure of Hydra offers a number of capabilities completely absent in the ownership protection structures of operating systems such as TOPSTEN, MULTICS, and TENEX. To attempt to exhibit and assess the contribution of these added capabilities a user-level file system (Almes & Robertson, 1978) on C.mmp was produced with significant enhancements due to the protection system flexibility. In addition, a secure mail system is under construction.

Other tasks have been undertaken to test the flexibility of hydra. Zog, a research project in man-machine communicaton has been brought up on C.mmp (Newell et al, 1977). Its required response objective of frame display in less than 1/10 second has been achieved under the Hydra operating system. The Harpy speech recognition system was redesigned as a decomposable program and successfully executed on Hydra. Algol 68 is currently running as a Hydra subsystem. Parallel processing primitives were designed and implemented.

5.5.6 Alphard

The Alphard project is currently in progress with major funding provided by NSF. A complete design was finished September 77 (Hilfinger et al, 1978). The implmentation of the language is in progress. The investigation into the implementation of high level languages such as Alphard has led to the discovery of new issues concerning types and genetics. Solutions to these problems and the dependence on the availability of PQCC technology has delayed implementation (Wulf et al, 1976). Nevertheless, a substantial effort has gone into testing Alphard's design resulting in contributions in the areas of verification and implementaton. A number of different programs have been constructed addressing the problem of defining control abstractions, verification methodology (Shaw et al, 1977).

5.5.7 L*: system implementation system

The analysis of L* as a system implmentation system was approached in two ways during the contract period. First, experience was gained through the construction of a set of large complex systems in the L* environment. These systems were Zog (Newell, et al., 1977), funded by CNR, HSP (McCracken, 1977), and Hearsay-C. The second approach was an investigation into new methods for evaluating large systems. The resulting analysis techniques, though incomplete in an experimental sense, were tested on the L* system. Applications of the analysis techniques and experience with building large systems, uncovered trends that L*-like systems increase productivity drastically.

The implementation of L* on CM* received low priority during the contract period. This was due to the decision to build software on CM* in Bliss rather than L*. Hence, L* has not been implemented on CM* at this time.

Experiments were run, testing the portability of L* to different machines. L* was implemented on the 8080 and Alto (funded by Xerox). In each case a minimal system (similar to that on the PDP-10) took one man/month to implement.

5.5.8 Algol 68

The implementation of Algol 68 with full runtime environment and multi-processing primitives was completed by July 1976. Algol 68 source code compiles into a machine independent the virtual-machine code. Runtime environments for Algol 68 currently exist on C.mmp and CM*. A run-time system for the PDP-10 will be available by June 1980. The portability of virtual-machine runtime environment demonstrates its machine independence.

The multiprocess pimitives in the C.mmp implementation of Algol 68 provide a general scheme for how to exploit miltiprocessing algebraic languages. The implementation of "eventual values" (Hibbard et al., 1977) allows the user and the system to designate operations to be separately scheduled when necessary. Experiments have been run testing the efficiency of these primitives as a

method of deriving parallelism. More exprimentation is needed to fully evaluate them.

5.6 Annotated Bibliography

This section contains a partial list of reports published in the area of Software Technology. An abstract accompanies each.

Almes, G. and Robertson, G. An extensible file system for Hydra. Proceedings of the Third International Conference on Software Engineering, Atlanta, GA, May, 1978.

An extensible file system has been designed and implemented for Hydra, an advanced capability-based operating system. This system demonstrates three notable advances to subsystem design: It provides a protected and efficient implementation via user-level code of functions ordinarily implemented as part of a conventional system's monolithic privileged section, It provides practical solutions to two protection problems, the Modification Problem and the Confinement Problem, for users of the file system, and It provides separation of mechanisms for data representation from mechanisms for protection and synchronization, thus allowing an extensible family of subfile systems to evolve. This paper treats the design and implementation of the Hydra File System and reflects on its implications for subsystem design and implementation.

Brown, K. Q. Fast intersection of half spaces. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

The problem of intersecting N half-spaces in K space is transformed to 2*K problems of constructing the convex hull of N points in K space and a simple intersection problem. This enables one to intersect the N K-dimensional half spaces in O(K*H(N,K)) time, where H(N,K) is the time required to construct the convex hull of N points in K space. For two and three dimensions the algorithm takes O(N log N) time in the worst case, but under fairly robust conditions the expected time is only O(N). It is also shown that an algorithm for intersection of half spaces can be used to construct the convex hull of points in K space. Thus, the intersection of half spaces and convex hull of points problems are essentially equivalent.

Cattell, R. G. A survey and critique of some models of code generation. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, November, 1977.

Various work on code generation is discussed, particularly from the point of view of simplifying and/or automating the derivation of this phase of compilers. Code generators,

which typically translate an intermediate notation into target machine code in one or more steps, have been relatively ad hoc as compared to the first phase of compilers, which translates a source language into the intermediate notation. Progress in formalizing the code generation process is summarized, with the conclusion that considerably more work remains. Future directions of research are suggested.

Cattell, R. G. Formalization and automatic derivation of code generators. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, April, 1978.

This work is concerned with automatic derivation of code generators, which translate a parse-tree-like representation of programs into sequences of instructions for a computer defined by a machine description. In pursuing this goal, the following are presented: A model of machines and a notation for their description, a model of code generation and its use in optimizing compilers, an axiom system of tree equivalences, and an algorithm for derivattion of translators based on tree transformattions (this is the main work of the thesis). The algorithms and representations are implemented to demonstrate their practicality as a means for generation of code generators.

Cohen, E. Problems, mechanisms and solutions. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, August, 1976.

This thesis formalizes the notions: problem, mechanism and solution, and shows how such a formalization is useful in describing problems and proving the correctness of solutions to them in computational systems. Mechanisms are formally defined as mappings (layers) between two computational systems. They provide natural models for protection, synchronization, and sequential and parallel control mechanisms. Certain algebraic properties of mechanisms are discussed; these correspond to properties one would ordinarily consider in studying the mechanisms listed above. We consider those problems in computational systems that may be solved either by adding a mechanism to a system or by imposing a constraint on the states in which the system is initially permitted to operate. We find that many such problems can be described as behavioral problems, constraints on the behavior of a system. These problems may be described in a manner that is independent of the particular system. A variety of important protection problems are defined in this way. We develop a formal methodology for solving problems in systems with multiple mechanisms. We use it in developing a number of solutions

to a particular protection problem, which we solve by constraining both a protection mechanism (determining acceptable initial protection configurations) and a control mechanism (specifying properties that must be satisfied by programs which are to be executed by certain users). Finally the thesis develops a variety of constructs for specifying behavioral problems and discusses considerations for analyzing, comparing and measuring solutions to them.

Cohen, E. Strong dependency: A formalism for describing information transmission in computational systems. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, August, 1976.

This paper presents an information theoretic approach to information transmission in computational systems. We formalize the effect of constraint on information paths and develop a number of inductive techniques for proving the absence of information transmission. Finally, we show how ordinary inductive assertions can be used in conjunction with the theory to analyze information paths in sequential programs.

Feiler, P. Evaluation of the bitstring algorithm. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, April, 1978.

A resource allocation technique based on an alternative data representation to the list structure, i.e. the bitvector, is discussed in this paper. The data structure provides for implicit collapsing of available resources, and the algorithm, called Bitstring, can be applied to any type of resource without performance loss. An optimized implementation of Bitstring is compared with a corresponding list structure algorithm (Firstfit). Two bitvector algorithms for special resource allocation environments, Exactstring and Quickstring, are presented. The implementation of Bitstring in microcode on a PDP-11/40E and the resulting performance improvement relative to the assembly code implementation are discussed.

Flon, L. and Suzuki, N. Nondeterminism and the correctness of parallel programs. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, May, 1977.

We present weakest pre-conditions which describe weak correctness, blocking, deadlock, and starvation for nondeterministic programs. A procedure for converting parallel programs to nondeterministic programs is described, and the correctness of various example parallel programs is treated in this manner. Among these are a

busy-wait mutual exclusion scheme, and the problem of the Five Dining Philosophers.

Flon, L. On the design and verification of operating systems. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1977.

This thesis applies and extends mathematical program verification to systems programs. The thesis is both methodological (in proposing a methodology for the design and verification of large programs), and theoretical (in presenting various results dealing with the correctness of parallel programs). The design methodology is based upon the use of abstract data types and the construction and verification of both specifications and implementations for them. The abstract data type is a means of modularization which encapsulates the representation of a data structure and the algorithms which operate directly upon it. The specification technique appeals to various mathematical structures (e.g., sets and sequences) to describe an abstract state for objects of a given type. The correctness of the formal specificaaons is cast in terms of the proof of certain invariant properties of the abstract state. An axiomatic proof rule is given to formulate the theorems necessary for proving the invariance of predicates across formal specifications. The applicability of the methodology to operating systems is explored. It is found that a hierarchical decomposition is most amenable to verification, and that the implementation language used is a function of that hierarchy. The example of a process dispatcher module of a hypothetical operating system is used to illustrate the process of design, specification, implementation, and verification using the methodology. Various properties are proven of the abstract specifications, including one representation of the concept of fair service. Programs are then written for the specifications and their correctness is verified. Three different approaches to the total correctness of parallel programs are treated. The first uses the weakest pre-condition concept to explore statically the combinatoric interactions which may occur among parallel programs during execution. The method is complete but computationally complex. A second approach extends the axiomatic weak correctness results of Owicki to include a technique for proof of loop termination. The concept of a steady state loop invariant is introduced and used to establish the total correctness of an old scheme of mutual exclusion which appeals only to the indivisibility of memory access for synchronization. The third approach treats a syntactically restricted class of parallel programs. For this class we give definitions for the weakest pre-

conditions which guarantee weak correctness and absence of blocking, deadlock, and starvation. We also formulate theorems which use invariant assertions to circumvent the actual weakest pre-condition computation.

Greer, K. SPACS graphics editor. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, September, 1976.

SPACS is an interactive graphics editor for use with a stylus/ tablet input device in conjunction with a graphics display terminal. SPACS is well suited for making tables, flow charts, logic diagrams, and other similar schematic diagrams. The user may ultimately obtain a hard copy of her work via the Xerox Graphic Processor (XGP). SPACS is composed of a large PDP-11 program where the picture processing is performed, and a SAIL program on the PDP-10, which acts as an I/O link for saving and retrieving files. In addition there is another SAIL program for creating image files for the XGP.

Guarino, L. R. The evolution of abstraction in programming languages. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, May, 1978.

As our understanding of the role of abstraction in programming has improved, programming languages have evolved in their use and support of abstraction. This paper defines abstraction and discusses how the use of abstraction in programming languages assists the programmer. It traces in depth the development of support for the abstraction of objects and for the abstraction of control constructs in programming languages.

Habermann, A.N. On the concurrency of parallel processes. In Perspectives on Computer Science. (Jones, A. K., Ed.) Academic Press, New York, NY, 1977, pp. 77-90.

This chapter contributes to the 10th anniversary symposium of the Computer Science Department at Carnegie-Mellon University by looking at two old problems in concurrent processes in a new light. The first problem is that of restricting the order in which concurrent processes may operate on hsared objects. We show how such restrictions can be stated as path expressions. The second problem is that of deadlock states. An O(mn log n) deadlock detection algorithm is presented, which is based on the heapsort algorithm.

Habermann, A. N., Feiler, P., Flon, L., Guarino, L. R., Cooprider, L. and Schwanke, R. Modularization and hierarchy in a family of operating systems. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, February, 1978.

The objective of the "Family of Operating Systems" project has been to investigate the feasibility of constructing systems which use identical or similar resources and share basic design decisions. The concepts of "moule", or "address space" and "hierarchy" have been used with special care. Common to all family members is the virtual memory facility which controls dynamic address space transitions. Family members may differ in the facilities they provide in static address spaces. This report presents an overall description of the FAMOS system. Section 1 describes the basic ideas underlying the FAMOS system and Section 2 describes the implementation. A more detailed description is found in the official documentation of the FAMOS system. This documentation consists of a number of "module documents". Each module document comprises two parts, an introductory description which spcified the function and dependency of a module and a "type description" which defines the representation and implementation of a moule as static address space.

Hibbard, P. G., Knueven, P. and Leverett, B. Issues in the efficient implementation and use of multiprocessing in Algol 68. Proceedings of the Fifth Annual Conference on Design and Implementation of Algorithmic Languages, Rennes, France, May, 1977.

Two multiprocessor computer configurations now exist at Carnegie- Mellon University. The first of these (historically) is C.mmp a closely coupled system of 16 processors sharing up to 16 million words fo memory by way of a fully concurrent crosspoint switch. The second configuration is Cm* a network of 10-100 processors, each with a local memory and connections to allow access to the memories of the others. The processors in these systems are minicomputers (modified DEC PDP-11's), and both architectures favor programs which may be decomposed into relatively independent parallel activities each of which can execute on a single minicomputer, rather than long sequential programs accessing a large data base. Exprience has shown that this decomposition of tasks is neither straightforward nor obvious. Thus, although both C.mmp and Cm* possess the overall instruction rate and memory size of a powerful maxicomputer, the difficult problem of utilizing the multiprocessing capabilities of the machine threatens to reduce the computational power perceived by

the user to that of a fast, time-shared minicomputer. This paper describes a continuing expriment to provide the programmer with facilities which allow a program written in a high-level program- ming language to be executed efficiently on the two multiprocessor systems. Only in part is the goal to produce a general purpose pro- gramming language for use on these machines; the main interest is to explore a particular method of specifying and exploiting parallelism, with a view to incorporating appropriate facilities in firmware. We start by describing the computers and the language system available on them. Next, a language extension, eventual values, designed to allow the expression of parallelism to be simple and convenient, is presented. Then, we explain how the concept behind eventual values has been built into the abstract machine of the language system on Cm*, and describe some of the properties of that system. Finally, example programs are given to demonstrate that considerable parallelism can be obtained from sequential programs with little or no explicit specification of parallelism.

Hilfinger, P., Feldman, G., Fitzgerald, R., Kimura, I., London, R., Prasad, K., Prasad, V., Rosenberg, J., Shaw, M. and Wulf, W. A. An informal definition of Alphard. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, .

The authors and their colleagues have been exprimenting with a collection of ideas about programming languages for several years. Our goals included determining the extent to which languages could support contemporary programming methodology, could aid in the construction of verifiable programs, and, at the same time, could be a completely practical programming tool. In the context of that exploratory spirit it seemed inappropriate to rigidly bind decisions about the details of the language. Hence, although our explorations were carried out in a relatively uniform notation and published under the name "Alphard", there really never was an Alphard language. The astute reader of our previous publications will have noted, and probably will have been frustrated by, the fact that we felt completely free to change the notation from paper to paper as the needs of our exploration seemed to warrant. With this document we are breaking with our previous strategy. We are now defining a specific language which we expect to serve as the basis of our further research. In the future we do not intend to alter this language in the sme free manner as we have in the past. There are two reasons for this shift in strategy: First, although we didn't admit it, much of the language was frozen in our heads, and the minor differences that appeared in published

examples only served to confuse our readers. Second, and far more importantly, we believe that the premises on which all the "data abstraction" languages are based are untested in practice. We feel the need to gain exprience before we can proceed with any confidence to tackle the next set of exploratory questions. To gain that experience we need to freeze, and to implement, at least some portion of the language -- and that is what we are now doing. Since we expect to work in the context of the language defined here for some time to come, the language is extremely conservative. Our past experience has been that simultaneously achieving verifiability and efficiency is possible -- but delicate. Hence we have chosen to include only features whose implications we fully understand. For example, we have omitted features dealing with concurrency, exceptional- condition handling, and so on. We fully appreciate that these features will be needed in a "production" version of Alphard; they are omitted here

because they are still the subject of our research. The present version of this report carries the word "Preliminary" in its title; we hope to promptly circulate a second version of the report from which this word has been elided. Our purpose in circulating this first version is to solicit comment. We will deeply appreciate any and all critiques of both the language and its presen- tation. Such comments should be sent to Bill Wulf, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 15213.

Hilfinger, P., Feldman, G., Fitzgerald, R., Kimura, I., London, R., Prasad, K., Prasad, V., Rosenberg, J., Shaw, M. and Wulf, W. A. An informal definition of Alphard. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, .

The authors and their colleagues have been exprimenting with a collection of ideas about programming languages for several years. Our goals included determining the extent to which languages could support contemporary programming methodology, could aid in the construction of verifiable programs, and, at the same time, could be a completely practical programming tool. In the context of that exploratory spirit it seemed inappropriate to rigidly bind decisions about the details of the language. Hence, although our explorations were carried out in a relatively uniform notation and published under the name "Alphard", there really never was an Alphard language. The astute reader of our previous publications will have noted, and

probably will have been frustrated by, the fact that we felt completely free to change the notation from paper to paper as the needs of our exploration seemed to warrant. With this document we are breaking with our previous strategy. We are now defining a specific language which we expect to serve as the basis of our further research. In the future we do not intend to alter this language in the sme free manner as we have in the past. There are two reasons for this shift in strategy: First, although we didn't admit it, much of the language was frozen in our heads, and the minor differences that appeared in published examples only served to confuse our readers. Second, and far more importantly, we believe that the premises on which all the "data abstraction" languages are based are untested in practice. We feel the need to gain exprience before we can proceed with any confidence to tackle the next set of exploratory questions. To gain that experience we need to freeze, and to implement, at least some portion of the language -- and that is what we are now doing. Since we expect to work in the context of the language defined here for some time to come, the language is extremely conservative. Our past experience has been that simultaneously achieving verifiability and efficiency is possible -- but delicate. Hence we have chosen to include only features whose implications we fully understand. For example, we have omitted features dealing with concurrency, exceptional- condition handling, and so on. We fully appreciate that these features will be needed in a "production" version of Alphard; they are omitted here

because they are still the subject of our research. The present version of this report carries the word "Preliminary" in its title; we hope to promptly circulate a second version of the report from which this word has been elided. Our purpose in circulating this first version is to solicit comment. We will deeply appreciate any and all critiques of both the language and its presen- tation. Such comments should be sent to Bill Wulf, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 15213.

Kimura, I. Cheap production of Japanese documents, an experiment in programming methodology. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

This paper describes a small experiment in programming methodology. The problem is to do someting nice for the producton of Japanese documents in a given environment. The assumed environment is that of the Department of Computer Science, Carnegie-Mellon University (CMU). The experiment is done by a one-man team consisting of the

author. The process involves four factors: (1) preparing data, (2) finding the properties of the computing environment, (3) designing the user interface, and (4) actually writing a program. All these proceeds in parallel, and results in an inefficient but well-considered "mock-up", on which a more efficient production version can be based. The program, written in Snobol 4, accepts a sort of romanized Japanese. The output, printed on the Xerox Graphics Printer of CMU, makes mixed use of the hirakana and the katakana characters, but the kanji (Chinese characters) is excluded. At the focus of attention is how the general shape of the software is determined, i.e., requirement analysis in the broad sense. We try to support the developer's imagination. For this purpose we combine desciplined and undisciplined life-styles. Relations to the works of Sandewall, Kernighan and Plauger, and others are discussed. The first half of this paper also serves as a user's manual of the product.

Lanciaux, D. and Wulf, W. A. Supporting small objects in a capability system. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, February, 1978.

Methodologies and checking techniques have been proposed to improve software reliability. It has also been argued that capability mechanisms are the natural support for these techniques because they enhance modular decomposition and information hiding. However, there is a conflict between these observations; modular decomposition limits the possible recovery actions to the information that a module can access directly. Each module must rely upon the reliability of those that it uses. This paper presents a mechanism which allows recovery to be managed at any level in the system while satisfying the information hiding principle. It is based on a save-restore mechanism. In addition, primitives to define consistent states in the system are provided by the Kernel.

Lanciaux, D. and Wulf, W. A. Error recovery in capability systems. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

Methodologies and checking techniques have been proposed to improve software reliability. It has also been argued that capability mechanisms are the natural support for these techniques because they enhance modular decomposition and information hiding. However, there is a conflict between these observations; modular decomposition limits the possible recovery actions to the information that a module can access directly. Each module must rely upon the reliability of those that it uses. This paper presents a

mechanism which allows recovery to be managed at any level in the system while satisfying the informaton hiding principle. It is based on a save-restore mechanism. In addition, primitives to define consistent states in the system are provided by the Kernel.

London, R. L., Shaw, M. and Wulf, W. A. Abstraction and verification in Alphard: A symbol table example. Proceedings of IFIP Conference on Constructing Quality Software, North Holland, 1978.

The design of the Alphard programming language has been strongly influenced by ideas from the areas of programming methodology and formal program verification. In this paper we design, implement, and verify a general symbol table mechanism. This example is rich enough to allow us to illustrate the use as well as the definition of programmer-defined abstractions. The verification illustrates the power of the form to simplify proofs by providing strong specifications of such abstractions.

Mason, P. H. Design tools for evaluating multiprocessor programs. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, July, 1976.

An approach to designing programs for implementation in a multiple instruction stream-multiple data stream processing environment is presented. A program is modeled as a directed graph consisting of two types of nodes: processing nodes and linking nodes. Communication among nodes in the model is represented by message tokens. Each processing node is similar in form to a semi-Markov process. A simulaton of the operation of the model is nondeterministic, but is based on prescribed probabilistic choice functons. A system, called STEPPS, has been built in which a model can be described and evaluation tools can be used to manipulate and act upon a model to predict performance of a program decomposition. The design approach is to describe a multiprocessing program in terms of the modeling syste. The model is examined to determine some analytic attributes of the model. The analysis available determines (a) whether the model is well formed, (b) whether the model contains deadlocks, (c) predictions of steady state properties of each process. In addition, without much difficulty, analysis functions external to STEPPS may be included as needed by a program designer. Some analyses, that may be interesting, may be difficult to determine without resorting to smulaton. Therefore the

STEPPS system includes a model simulator with data collection facilities. The STEPPS data collection facilities include such measures as wait times and queue lengths. As in the case of analysis functions, STEPPS allows the inclusion of data collection facilities not originally provided by STEPPS. As a system is designed, alternate models can be examined; and based on an individual designer's choice of performance attributes, a model can be chosen on which to base the construction of a multiprocessor program. As more is learned about the real system parameters, the model can be tuned to more closely predict ultimate system performance. Several examples of communicating processes are modeled using STEPPS including pipeline processes, probabilistic processes, P/V synchronization, and reader/writer synchronization. Two experiments are presented as validation of the usefulness of the STEPPS tools. In the Bliss/11 experiment, the implications of restricting the numbers of available processors and using different scheduling algorithms were examined, and the effect of using alternate program structures was explored. In the Hearsay II experiment it

was shown that, when a multiprocess program under development is sufficiently instrumented, the STEPPS model and system can be used to help tune the program's structure. The use of the tools for predicting the performance of a multiprocessing program falls between purely analytic models, such a queueing theory or Petri-nets, and system simulations built in a general purpose simulaton language. The STEPPS system is presented as a new approach to designing multiprocessing programs.

Newcomer, J. and Reiner, A. The HYDRA users manual. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, August, 1977.

This manual is a descendant of the Hydra Songbook, which was conceived, written, and disseminated by Brian Reid to fill the void left by the lack of user documentation for Hydra. Like the Songbook, this document is intended to contain those things that are of general interest to the user community. The User's Manual is published in two parts. The first contains those chapters that are of interest to general users, such as the introductory material and the chapters on the Command Language. The second contains reference material on the various subsystems. Most users will find the first part sufficient for their needs.

Schwanke, R. Survey of scope issues in programming languages. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

In this paper we shall study scope issues in programming languages, from the standard binding techniqurs and philosophies of early languages, to the recent work in data encapsulation. First we will study the fundamental comcepts of binding, then see how they appeared in early languages. The scope problems in these languages made clear the need for additional program structuring tools, leading to the development of data encapsulation mechanisms. We shall study the scope properties of data capsules, and compare the encapsulation philosophies of several modern languages. We shall use the notion of abstract data types to study modern scope issues, and to survey recent advances in several scope-related areas. Finally we shall compare and contrast several languages, both old and new, by studying solutions in each of them to a common programming problem.

Shamos, Michael I. Computational geometry. Technical Report, Yale University, New Haven, CT, May, 1978.

This thesis is a study of the the computational aspects of geometry within the framework of analysis of algorithms. It develops the mathematical techniques that are necessary for the design of efficient algorithms and applies them to a wide variety of theoretical and practical problems. Particular attention is given to proving lower bounds on running time and to analyzing the average-case performance of geometric algorithms. The approach taken is to isolate a computa- tional example, that determining whether any two of N line segments in the plane overlap is an essential step in many intersection applications. An optimal algorithm for this problem, therefore, becomes an important geometric tool that can be used to build other, more complicated, fast algorithms. This method is employed in a unified attack on the problem of the convex hull, various geometric search problems, finding the intersection of objects and questions involving the proximity of points in the plane. What emerges is a modern, coherent discipline that is successful at merging classical geometry with computational complexity. Among the major new results presented are a convex hull algorithm with expected running time that is linear in the number of input points, an O(N log N) algorithm for linear programming in two variables (which is superior to the Simplex method), and an O(N log N) algorithm for constructing a minimum spanning tree on a finite set of points in the plane.

Shaw, M. Abstraction and verification in Alphard: Design and verification of a tree handler. Proceedings of the Fifth Texas Conference on Computing Systems, Austin, TX, 1976. Also CMU-CSD

Technical Report June, 1976.

The design of the Alphard programming language has been strongly influenced by ideas from the areas of programming methodology and formal program verification. The interaction of these ideas and their influence on Alphard are described by developing a nontrivial example, a program for manipulating the parse tree of an arithmetic expression.

Shaw, M., Wulf, W. A. and London, R. L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. Comm.ACM 20, 8 (August 1977). Also CMU-CSD Technical Report August, 1976.

The Alphard "form" provides the programmer with a great deal of control over the implementation of abstract data types. In this paper the abstraction techniques are extended from simple data representation and function definition to the iteration statement, the most important point of interaction between data and the control structure of the language itself. A means of specializing Alphard's loops to operate on abstract entities without explicit dependence on the representation of those entities is introduced. Specification and verification techniques that allow the properties of the generators for such iterations to be expressed in the form of proof rules are developed. Results are obtained that for common special cases of these loops are essentially identical to the corresponding constructs in other languages. A means of showing that a generator will terminate is also provided.

Shaw, M., Almes, G., Newcomer, J., Reid, B. K. and Wulf, W. A. A comparison of programming languages for software engineering. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, April, 1978.

This report is a comparison of four programming languages in the light of modern ideas of good softtware engineering practice. We include three existing languages (Fortran, Cobol, Jovial) and the proposed DoD standard for programming embedded computer systems. At least some of tthe acrimony that surrounds the discussion of the relative merits of various programming languages arises from a failure to draw a proper distinction between comparison and evaluation. In a comparison, the analysis is directed at distinctions and similarities of the languages, and possibly at the relative merits of particular features under certain circumstances. An evaluation, on the other hand, is aimed at making a judgmentt, hopefully in terms of a set of clearly-defined objectives. We wish to emphasize

that our primary goal in this report is a comparison of languages. Our comparison focuses on those dimensions that bear on modern notions of program structure and good software engineering practices.

Shaw, M., Hilfinger, P. and Wulf, W. A. TARTAN language design for the Ironman requirement: Notes and examples. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

The TARTAN language was designed as an experiment to see whether the Ironman requirement for a common high-order programming language could be satisfied by an extremely simple language. The result, TARTAN substantially meets the Ironman requirement. We believe it is substantially simpler than the four designs that were done in the first phase of the DOD-1 effort. The language definition appears in a companion report; this report provides a more expository discussion of some of the language's features, some examples of its use, and a discussion of some facilities that could enhance the basic design at relatively little cost.

Shaw, M., Hilfinger, P. and Wulf, W. A. TARTAN language design for the Ironman requirement: Reference Manual. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

TARTAN is an experiment in language design. The goal was to determine whether a "simple" language could meet substantially all of the Ironman requirement for a common high-order programming language. We undertook this experiment because we believed that all the designs done in the first phase of the DOD effort were too large and too complex. We saw that complexity as a serious failure of the designs; excess complexity in a programming language can interfere with its use, even to the extent that any beneficial properties are of little consequence. We wanted to find out whether the requirements inherently lead to such complexity or whether a substantially simpler language would suffice. Three ground rules drove the experiment. First, no more than two months -- AApril 1 to May 31 -- would be devoted to the project. Second, the language would meet all the Ironman requirements except for a few points at which it would anticipate Steelman requirements. Further, the language would contain no extra features unless they resulted in a simpler language. Third, simplicity would be the overriding objective. The resulting

language, TARTAN, is based on all available informaton, including the designs already produced. The language definiton is presented here; a companion report provides an overview of the language, a number of examples, and more expository explanations of some of the language features. We believe that TARTAN is a substantial improvement over the earlier designs, particularly in its simplicity. There is, of course, no objective measure of simplicity, but the syntax, the size of the definition, and the number of concepts required are all smaller in TARTAN. Moreover, TARTAN substantially meets all of the Ironman requirement. (The exceptions lie in a few places where we anticipated Steelman requirements and where details are still missing from this report.) Thus, we believe that a simple language can meet the Ironman requirement. TARTAN is an existence proof of that. We must emphasize again that this effort is an experiment, not an attempt to compete with DOD contractors. TARTAN is, however, an open challenge to the Phase II contractors: The language can be at least this simple! Can you do better?

Suzuki, N. and Jefferson, D. Verification decidability of Presburger array programs. Proceedings of the Conference on Theoretical Computer Science, Waterloo, Ontario, Canada, August, 1977.

A program annotated with inductive assertions is said to be verification decidable if all of the verification conditions generated from the program and assertions are formulas in a decidable theory. We define a theory, which we call Presburger array theory, containing two logical sorts: integer and array-of-integer. Addition, substraction, and comparisons are permitted for integers. We allow array contents and assign functions, and, since the elements of the arrays are integers, array accesses may be nested. The first result is that the validity of unquantified formulas in Presburger array theory is decidable, yet quantified formulas in general are undecidable. We also show that, with certain restrictions, we can add a new predicate Perm(M,N)--meaning array M is a permutation of array N--to the assertion language and still have a solvable decision problem for verification conditions generated from unquantified assertions. The significance of this result is that almost all known sorting programs, when annotated with inductive assertions for proving that the output is a permutation of the input, are verification decidable.

Wulf, W. A., Levin, R. and Pierson, C. Overview of the Hydra operating system development. Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, TX, November, 1975.

An overview of the hardware and philosophic context in which the HYDRA design was done is discussed the construction methodology is discussed together with some data which suggests the success of this methodological approach.

Wulf, W. A., London, R. L. and Shaw, M. An introduction to the construction and verification of Alphard programs. IEEE Trans. on Software Engineering SE-2, 4 (December 1976).

The programming language Alphard is designed to provide support for both the methodologies of "well-structured" programming and the techniques of formal program verification. Language constructs allow a programmer to isolate an abstraction, spedifying its behavior publicly while localizing knowledge about its implementation. The verification behaves in accordance with its public specifications; the abstraction can then be used with confidence in constructing other programs, and the verification of that use employs only the public specifications. This paper introduces Alphard by developing and verifying a data structure definition and a program that uses it. It shows how each language construct contributes to the development of the abstraction and discusses the way the language design and the verification methodology were tailored to each other. It serves not only as an introduction to Alphard, but also as an example of the symbiosis between verification and methodology in language design. The strategy of program structuring, illustrated for Alphard, is also applicable to most of the "data abstrac- tion" mechanisms now appearing.

Wulf, W. A. and Harbison, S. P. Reflections in a pool of processors: An experience report on C.mmp/Hydra. Proceedings of the National Computer Conference, Anaheim, CA, June, 1978. Also CMU-CSD Technical Report February, 1978.

This paper is a frankly subjective reflection on the successes and failures of the C.mmp project by those most intimately connected with its design, implementation, and use. It attempts to catalog and characterize the things we feel we did right and the things we did wrong. We sincerely hope that this sort of evaluation will help others who undertake similar projects.

6. MULTIPLE PROCESSOR SYSTEMS (MPS)

Multiple Processor Systems (MPS): The major question of interest is: What types of computer systems composed of multiple processors are cost-effective and reliable? The primary concern is exploring the space of multiple processor systems by creating experimental systems (design, construction, and implementation of both hardware and system software) and analyzing their performance and behavior. The current foci are on exploiting C.mmp, a 16x16 multi-miniprocessor which is now operational for users, and on implementing Cm*, a 50 Pc multiprocessor built with LSI-11s.

6.1 Introduction

We proposed to explore the space of multiprocessing architectures with respect to cost-effectiveness, total power, reliability and modularity; and to do this by constructing total multiprocessing systems (hardware, operating software, user systems) so that quantitative answers may be obtained to central questions of the performance of these systems on real problems.

Specifically, with respect to C.mmp, we : (1) turned the total system over to Operations, thus completing its basic development; (2) continued our experimental performance analyses and constructed models of the total system behavior; (3) initiated an effort to obtain software reliability, working with a predictive model of total reliability; and (4) continued developing user applications to provide the basic data for evaluation of C.mmp.

Specifically, with respect to Cm*, we: (1) brought into operation the 10 processor system; (2) began implementation of the 50 processor system; (3) created operating software on the system; (4) carried out a basic cost/performance analysis; and (5) developed alternative multicomputer schemes representing other interesting points in the design space.

6.2 Background: Pre-Contract Period

A multiprocessor is a multiple computer system in which the several processors share common primary memory (in contrast to a network, in which the several computers do not share memory, but rather communicate through relatively narrow bandwidth communication channels).

For a given state of technology there are two strong reasons for considering a multiple computer structure, and a multiprocessor in particular. First, for large N it may be technologically infeasible to build a uniprocessor which is N times more powerful than available, nominally "large", uniprocessors. Second, primarily because of the advances in LSI, a small processor of 1/Nth the power costs far less than 1/Nth of the nominal "large" processor. Thus the cost/performance ratio strongly favors multiple small-processor systems.

In principle, at least, there are a number of additional reasons for considering multiprocessor systems. The more important are (1) reliability and graceful degradation, and (2) incremental expansion. A system containing N identical processors should have the property that the failure of a single processor merely results in the loss of 1/Nth of the processing power. A multiple processor system should also have the property that additional processors (and memory) can be added in response to increased demand. Thus an installation may "start small" and increase its processing power only as needed.

The potential advantages of multiprocessor systems have been known for a long time, and examples of them (with relatively small numbers of processors) date from the early 60's. Yet there have been a few multiprocessors with more than 2 processors and the amount of scientific knowledge available about them is miniscule. There are many reasons for this lack of both examples and knowledge, including the prohibitive cost of experimentation in the pre-minicomputer era. The advent of the minicomputer, and more recently the microcomputer, however, has made it economically feasible to conduct research

on these structures, and a number of Universities and industrial research organizations have done so.

The major problems to be faced, and scientific results obtained, relate to (1) positing a hardware structure which realizes the inherent cost/performance and modularity advantages of these structures, and (2) formulating software structures (operating systems, languages, and applications) which realize these hardware advantages. The space of design alternatives is extremely rich, and any given hardware structure can only be used to explore a portion of it. Similarly, within a given hardware structure only certain software structures make sense.

The long range goal of the CMU MPS program is to explore all of the technical/scientific problems associated with multiprocessor systems, and to do so in a manner which maximizes the amount of scientific knowledge made generally available. To achieve this requires (1) attention to all aspects of the construction of individual systems, (2) the construction of more than one system -- each exploring a different portion of the design space, and (3) a theoretical analysis of design space itself.

By "attention to all aspects" of the construction of a multiprocessor system we mean a systematic scientific investigation of at least the following issues: (1) the architecture; (2) the operating system; (3) the software facilities; (4) task (application) decomposition and analysis; (5) system performance; and (6) total system reliability. At present we are exploring all these issues in two distinct regions of the design space with C.mmp and Cm*.

C.mmp is a canonical multiprocessor in which up to 16 processors (various models of the PDP-11) have access to a large shared memory (up to 32 million bytes) through a central switch. The time to access any particular memory location is uniform and hence one expects higher performance than with other multiprocessor structures; expansion is limited by the physical structure of the switch and the switch might become a reliability bottleneck. A more

detailed description of C.mmp may be found in 1975-76 CMU ARPA Proposal.

Cm* is an alternative structure in which each processor has a limited amount of local memory (28K words); processors may access memory of other processors through a shared, multi-level bus structure. The processors are microprocessor implementations of the PDP-11, called the LSI-11. This structure permits much more flexible expansion (unlimited in principle) and avoids the reliability concerns of a centralized switch. However, since access to remote memory is more expensive than to local memory, performance is a strong function of the degree to which programs can be organized to make predominantly local references. The immediate goals of the two systems are similar in that they relate to progressing toward the long-term goal; they differ so far as they reflect the relative maturity of the two systems.

C.mmp is a relatively mature system: a full 16-processor, 2.2 megabyte system became available in Summer 77, the operating system (Hydra) is quite stable. A number of programming systems (L*, Algol 68, etc.) and performance evaluation tools are finished. Thus the immediate goals for C.mmp relate to testing the hypotheses of its design -- that is, answering the scientific questions implicit in the above list.

Cm*, on the other hand, is a new effort, funding having started in the 75-76 contract year. Although a small prototype (10 processers) has been constructed, the present goals relate primarily to bring it to the state of maturity at which the scientific questions can be posed. (It should be noted, however, that many of the tools constructed for C.mmp (hardware monitor, script driver, Bliss, etc.) can be used with Cm*, thus we expect more rapid progress toward maturity than was possible with C.mmp.)

6.3 Relevance of Research

The MPS program does not focus on a restricted task domain. Rather, its results are aimed at all applications in which reduced cost, increased performance, and reliable and expandable computing power are desired. The trend of technology is unmistakably toward mass production of increasingly

sophisticated elements on a single chip; the major question is how to best use this technology to achieve these goals. This is precisely the target of the MPS program.

Our goal is to explore the multicomputer design space, and to understand at least some of the most promising points within it in depth. To that end both of our current major systems, C.mmp and Cm*, have performance analysis as a major component. We are not content to simply demonstrate that a particular architecture can be built, we demand (of ourselves) that its properties be quantified. This is especially important for multiprocessor systems, since the size of the gain is of the essence for applied problems.

Bringing a system to the point at which meaningful evaluation may be performed requires attention to all aspects of the system: hardware, operating system, languages and other tools, and task decomposition. C.mmp is at that state in almost all essential ways; Cm* is not. Thus the relevance of the effort on C.mmp is more immediate: It relates to answering essential scientific questions about the canonical multiprocessor structure. The relevance of the efforts on Cm* is more indirect. It relates to constructing a system on which these same questions can be asked in a few years.

The list of scientific questions to be asked about any multiprocessor must include at least the following:

- Cost/performance: The intuitive rationale for the superiority of the cost/benefit ratio is compelling. However, we must ask whether it has been realized for real applications under real loads (or whether the potential advantage has been lost to complex hardware and software interactions in controlling the assemblage of processors).

- Problem decomposition: For a single program to exploit the potential parallelism, it must be decomposed into concurrent subtasks. If no natural decomposition corresponding to the structure of the architecture exists, the total power of the system cannot be realized for that task. If this were true for many (most) tasks, the utility of the architecture is seriously in doubt.

- Reliability: The intuitive rationale for the inherent reliability of multiprocessors is compelling, but we must ask whether it has been realized. The hardware/software structure might be so complex

that the potential reliability gains are lost or are bought at the price of too much of the potential power gain.

The answers to these questions will provide basic input to the design of the next generation of multiprocessors, specifically those produced for military use. Though multiprocessors have been built for the military (and some go back quite a few years), it is discouraging to try to identify specific knowledge gained from these systems that is generally available for the design of new systems (especially the sort of quantitative analyses we will be carrying out on C.mmp and Cm*). The absolute practical importance of building up the fund of scientific, verified knowledge on multiprocessor systems simply cannot be overestimated.

We also assert that the artifacts themselves (both hardware and software) are highly relevant deliverables. Both C.mmp and Cm* could be replicated (we have discussed several possibilities for this in connection with C.mmp and Cm*). The operating system for C.mmp, Hydra, could be exported as the basis, for example, of a highly secure system. We expect the same to be true for Cm*. Some of the applications being developed for C.mmp could also be exported.

We now turn to the relevance of the specific efforts proposed:

The C.mmp configuration: C.mmp is a "canonical" multiprocessor, ie, one which provides complete interconnection between all processors and all memory ports. It is the configuration that provides the best possibilities for obtaining general computing power, adaptable to a wide array of uses, since the access cost is uniform and thus all processors and memory can be treated alike by the algorithms and operating system. It constitutes a fundamental point in the multiprocessor design space.

The particular C.mmp design is an important configuration: (1) providing a larger number of processors (16) than is elsewhere available, and (2) having its processors dynamically microcodable (the 11/40E), which permits a much wider range of adaptation in exploiting the potentially available raw power.

C.mmp has become a unique resource, available to a wide community of users (it is available over the ARPA Net).

Performance Evaluation: One fundamental class of questions about multiprocessor systems relates to their performance on a single program: what degree of parallelism is possible on a given class of problems, and what sorts of overhead are paid for coordinating and synchronizing the program. These are answered by experimenting with specific benchmark programs with known properties.

A second fundamental class of questions relates to the effects of variation in the total computing environment in which one programs. Here one normally depends on measuring and analyzing performance with the variety of loads that happen along. More generally applicable results are obtained by using script drivers (as we do) to create arbitrary (artificial) computing environments. Extrapolation beyond the particularities of the C.mmp configuration depends on constructing validated models. These activities represent the culmination of a major portion of the work on C.mmp and are directly responsible for much of its applied relevance.

Reliability: A multiprocessor hardware configuration presents an occasion for reliable operation. To obtain it one must create a large number of software mechanisms. For this to be relevant to obtaining multiprocessors systems useful in applied contexts one must not only create the mechanisms, but measure them against a theoretical model that permits the knowledge to be extrapolated. In fact, the actual software mechanisms will have general applicability, though this will need to be tested.

Other Scientific Activities Related to C.mmp: The relevance of multiprocessor research cannot be divorced from that of the applications which run on the multiprocessor -- if for no other reason than that the applications provide the bases for performance evaluation, especially of complex computational requirements. Some of the applications on C.mmp are directly related to problems of applied interest. The Blackboard Model for Hearsay has

already been discussed. Another one is a study being conducted for DCA (Defense Communications Agency) on the use of multiprocessors for communication functions.

10 and 50 Processor Cm* Systems: Cm* represents a particular form of indefinitely extendable multiprocessor. Classical multiprocessor architecture (ie, C.mmp and its kin) reaches a barrier in the size of the crosspoint switch, and in the need to fix its size at design time (with current switch designs). A 1000*1000 classical multiprocessor requires 10^6 cross points. More flexible multiprocessor architectures build the switch in a modular fashion, depending on a configuration that does not require all processors to be connected directly to all memories. The price paid for this is forced locality of memory access, ie, significantly different times are taken to obtain data from different memories (more time from those farther away in the connective structure). This significantly complicates the (programming and algorithm design) task of getting efficient computing, perhaps even restricting the task domain of the system. But they permit more computers to be incorporated into a single structure.

The Cm* interconnection strategy is dictated by current technology and is a cluster scheme; that is, a number of processors share a multiplexed interprocessor bus and relocation unit (Kmap). This group of processors, together with the bus and Kmap, is called a cluster. Multiple clusters may be interconnected by an "intercluster" bus.

Software: As noted in an earlier section, while it may not appear that software development is in the mainline of multiprocessor research, in fact extensive software is needed and needed rapidly. Moreover, obtaining appropriate software is a strong condition of applicability of multiprocessor systems. Especially in the early years of multiprocessor development, before any significant standardization of architectures has occurred, this will be a major problem to be faced by the military. The general techniques we use (higher-order languages (Bliss and L*), use of a larger processor (the PDP-10)

for support, adaption of programs written elsewhere, etc.) are not unique (though our emphasis on measuring our software production performance is unfortunately not wide spread). Nevertheless, the tools we have developed are distinct contributions to the construction of large system and are available for other applications in DOD and elsewhere.

Analysis of CM*: The issue here is essentially the same as for C.mmp. The applicability of Cm* depends in large measure on what it contributes to our understanding of the space of multiprocessor designs; and this in turn depends on how well we carry out detailed quantitative measures and use them to create models that permit us to generalize about a region of design space.

Alternative Systems to Permit Evaluation: As already stressed, the relevance of our work on MPS depends in part on our development of an understanding of the space of multiprocessors. This is what gives the maximum information to the design of multiprocessor systems for the military and others. We can construct only a limited number of full systems, but we can explore other variants in more limited ways. The flexibility in both C.mmp and Cm* permit this.

A prime example of the usefulness of exploring several alternatives is constructing and experimenting with a network of LSI-11s. Networks of computers (computers interconnected by channels that transmit messages between primary memory, rather than permit common direct access) are the type of multiple computer system being investigated most intensively in the computer science community. Networks form another basic point in the design space, and comparing C.mmp and Cm* to equivalent network structures will increase the applicability of the knowledge about the design space we are generating.

6.4 Accomplishments: Pre-Contract Period

Although CMU's involvement with multiple computer systems dates from the G-21 (a two-processor multiprocessor) in the early 1960's, the relevant history dates from two events in the early 70's. The first of these was a study sponsored by ARPA into the nature of the appropriate next generation

computer for AI (here called MI). This study was performed by a committee of representatives from several APRA sites; its final recommendation was the design of a multiprocessor system called C.ai (which was not built). The second was a CMU thesis by Strecker which provided an analytic model of memory contention in a multiprocessor. The first event provided the impetus to study multiprocessors in greater detail, the second provided the ability to design a relatively large multiprocessor with some assurance that its performance would not be unduely degraded by contention for the shared memory. The specific result of these two events, in the context of rapidly decreasing cost/performance of minicomputers, was the decision to construct C.mmp.

Two other events also relate to the present shape of the CMU multiprocessor program. One was the participation of Bell and Wulf in the design of the PDP-11; both were members of the original design team. The second was the development at CMU of RTM's (register- transfer modules, similar in intent to Clark's macro-modules). This work was supported by NSF, and RTM's later appeared commercially as the PDP-16. RTM's were the precursor of computer-modules, and hence of Cm*.

The following discussion of more recent accomplishments is somewhat arbitrarily divided into hardware, operating system, and application topics. It should be remembered that, despite the division, we have a strong commitment to viewing these as components of a uniform goal; thus specific achievements are considered as instances of progress toward this goal. The nature of large system construction is such that the major scientific hypotheses can be fully tested only when the system has reached a certain level of maturity; until that point one can reach only tentative conclusions. (Which is not to say that significant contributions cannot be made along the way. Indeed one expects that such contributions will be made).

C.mmp has reached the level of maturity at which meaningful tests can and have been made, and experiments on its performance, flexibility of its operating system, etc. are now in progress. Results from these experiments

are described in this report, but we also expect to continue to perform experiments for several more years. Cm* is at an earlier state, having only started during the current contract year. It is still in the design and construction phase, and it is far too early to be subjected to extensive test (although analytic models have been constructed and preliminary performance comparisons against the design have been made).

C.mmp (hardware): We have constructed a non-trivial multiprocessor; the current configuration includes 5 PDP-11/20 and 4 PDP-11/40 processors, 1.2 megabytes of memory, and a wide variety of peripheral devices. Preliminary indications are that the memory contention is within predicted bounds (although the nature of the performance curve is such that final confirmation must await a larger configuration). The remaining 7 processors have been purchased, as has an additional 1.0 megabytes of memory; and have been connected. Reliability of the major components has been good (eg, no known switch errors in the past several months), although overall system reliability has been initially rather poor (primarily due to the addition of new processors each time a stable point is reached with the existing configuration), it has stabalized overtime.

Two other hardware developments relate to C.mmp, but also stand as significant achievements in their own right. The first is a programmable hardware monitor which is capable of being dynamically controlled by one PDP-11 while monitoring a second (which can, in fact, be the same as the controlling one). The second is the PDP-11/40E, a modification of the basic PDP-11/40 to include 1024 words of writable microstore (to calibrate this number, the basic PDP-11 instruction set requires 256 words of microcode). The modification also includes an upgrading of the functional capability of the microcode interpreter to include multi-bit shifts, masking, and a number of other useful features. All of the 11/40 processors attached to C.mmp are 11/40E's. A copy of the 11/40E has also been loaned to the Navy for its evaluation of instruction set designs (the CFA project). In addition, copies

of the PDP-11/40E's are in use at BBN, Naval Research Laboratory, and the Technical University of Berlin.

Most of the non-standard C.mmp hardware was constructed by the departmental engineering lab. This included: the switch, master clock, interprocessor interrupts, interval timers, the hardware monitor, and 11/40E.

Hydra: Some aspects of Hydra are covered under software technology, but here we list some achievements related specifically to the underlying multiprocessor structure. First, a distributed operating system has been constructed -- ie, one in which there is NOT a master-slave relation between the processors. Both the operating system and user programs view the processors as an anonymous pool, and any process (user or operating system) may execute on any processor at any time. Second, protection and resource policies have been removed from the operating system and delegated to user-level, non-privileged programs; thus experimentation along these two crucial dimensions of operating system variability are possible within the Hydra context. Third, nearly all facilities normally thought of as provided by the "operating system" are, in fact, defined, and definable, by user-level software; specialized facilities may be easily added by any user -- thus specializing the system to his task. Fourth, a capability-based protection structure has been provided which permits the solution of a greater range of protection and security problems than has hitherto been possible.

Support Programs: It is not the case that one may build a system such as C.mmp or Cm* without the wide-range of user support programs which one normally associates with a "general purpose" system. In order to obtain meaningful results on the architecture, operating system, etc., one must have users -- and users must have editors, debuggers, compilers, etc. Providing these facilities is not in the mainline of multiprocessor research, but they must be done, and we have chosen to make each (or most, anyway) into experiments in software technology. The facilities currently available include: a command language, file system, directory system, system status,

Bliss/11, L*, Algol 68, device allocation, multi-precision and floating point routines, formatted i/o, etc.

Performance Evaluation: The effort to date has largely been devoted to tool construction: a hardware monitor, script driver (which runs on a separate, front-end processor), event tracer in the operating system, and benchmark program development. At this stage we are prepared to perform a number of substantial experiments, and indeed, a number are in progress. Some small preliminary experiments have been performed which tend to confirm that: (1) memory contention is within expected bounds, and (2) the overheads imposed by Hydra for functions similar to those on conventional uniprocessor operating systems are comparable to what one finds on those systems. We are in the midst of a series of experiments (using the script driver) to test system response under various loads; for example, some VERY preliminary, unpublished results indicate that under moderately heavy loads (75% processor utilization) terminal response to the trivial request is less than 0.15 seconds. Detailed tracing of Hydra activity during these experiments is being used to tune system performance.

Applications programs: A number of applications programs have been developed for execution on C.mmp/Hydra. These include a multiprocessing version of Hearsay II (see 3.2), a multiprocessing image-processing program (Oleinick), a multiprocessing chess program (to explore parallel search algorithms), etc. Each such program is treated as both a study of multiprocess task decomposition and as a benchmark for the hardware/software system.

Cm*: As noted earlier, Cm* is not as mature as C.mmp, and is in the construction phase. Specifically, its state can be characterized as follows: (1) the architectural design is complete, (2) a number of simple analytic models have been built and are used to evaluate proposed architectural changes, (3) the interface to the interprocessor bus (Slocal) has been built and tested, (4) the interprocessor bus control (Kmap) has been designed and is

being fabricated, (5) a testbed system involving 3 LSI-11's and a PDP-11/10 host (used for monitoring and debugging) has been assembled, (6) software for the host has been written and is being used, (7) an emulator for Cm* has been written on C.mmp and will be used for software development, and (8) design of an operating system is in progress. In addition, much of the tool development for performance evaluation on C.mmp (the hardware monitor and script driver in particular) will be usable on Cm*.

6.5 Accomplishments: Contract Period 1976-78

C.mmp: Symmetric Multi-Mini-Processor System

6.5.1 16 Processor Configuration

C.mmp was completely configured by the Fall of 1976. It is composed of 16 processers: 12 microcoded PDP11/40E's and 4 PDP11/20's. 2.2 megabytes of primary memory can be accessed via the 16 x 16 crosspoint switch. C.mmp was made available as a regular facility in January 1978, and its performance since then has been both stable and reliable (Fuller and Harbison, 1978b, CMU-CS-78-146). A retrospective analysis of C.mmp/ Hydra can be found in (Wulf and Harbison, 1978a).

6.5.2 Performance Evaluation

Experiments were carried out to measure the affects of processor, memory, synchronization, and scheduling variations upon parallel algorithm execution (Oleinick and Fuller, 1978; Oleinick, 1979). It was found that variation in processor speeds greatly affect "staged" algorithms; code sharing and process schedulers are potential bottlenecks; and synchronization primitives may incur high overhead of process when time slices are small.

6.5.3 Reliability

An intensive study of hardware reliability was made on C.mmp (Siewiorek et al, 1978b). A probabilistic hard failure model was constructed. It is a modification of the MIL 217B model (Military Standardization Handbook 217B) based on data accumulated both at CMU and chip vendors. Substantial gains in reliability prediction have been attained through more accurate models of

systems and their failure modes. It was discovered that a small amount of redundancy improves system reliability, but more redundancy is ineffectual. Also, highly reliable components increase system reliability dramatically. The effect of system modularization on reliability is dependent on component usage.

Research has continued towards increasing system reliability through software mechanisms (Siewiorek et al, 1978a). Two types of errors have been dealt with: 1) frequent errors (transient) incurring local damage only and 2) rare errors resulting in system-wide damage. Errors of the first type, interprocessor interupt failures, DMA overruns, and memory parity failures were discovered and either automatically fixed by recovery code, or reported to the user, as in the case of parity errors.

For the second class of errors, a suspect/monitor model was devised. An errant processor (suspect) is placed under the control of another processor (monitor) who exercises the suspect using diagnostic routines. Subsequent suspect errors discovered during monitoring may result in a processor being reloaded, quiessed, or removed from the processor pool.

6.5.4 Other scientific activities relevant to C.mmp

Machine Intelligence systems have the potential for fully utilizing multiprocessor systems. Two experiments were run to test the feasability of C.mmp for supporting MI research.

The Hearsay-II system with simulated knowledge sources was implemented. Using tailored synchronization primitives, tests show an expected speed-up in recognition (Lesser & Fennell, 1977).

A production system version of the Hearsay-II syntax and semantics, and word hypothesization modules was designed and implemented on C.mmp (McCracken, 1978). It was found that a production system Hearsay (HSP) achieves a greater amount of parallelism at a lower cost than HSII. Thus greater utilization of processors is expected than in a typical decomposition of HSII for C.mmp.

Because of the distribution of productions in memory, hardware memory interference did not limit parallelism. It was also found that production system architectures are not limited by small address spaces because of their inherent modularity.

The following software systems have been constructed and are currently available:

1. Hydra: Operating sysem kernal.
2. L*: A high level symbol manipulation language for AI research.
3. Algol68: An algorithmic programming language providing multiprocessing primitives.
4. CM* Simulator: A software package for simulating CM*.

6.5.5 10 processor CM* system

The goal of constructing a single cluster system composed of 10 processors and 3 kmaps (address translation microcomputer) was achieved and demonstrated to ARPA (May 1978). Included was the demonstration of the StarOs operating system (Jones et al,1978).

6.5.6 50 processor CM* system

The decision was made to extend Cm* to 50 processors (LSI/11's supplied by DEC). The system is expected to be operational by December 1978.

6.5.7 CM* Software

The initial three software tasks for a new machine are (1) to produce a simulator so that work can proceed before the machine is operational; (2) to produce an operating system; and (3) to produce a basic set of software facilities. The latter two must come up as quickly as possible once the machine is available. Cm* is considered a highly experimental device. Thus the demands for these basic three are somewhat specialized, but they still exist.

A Cm* Simulator was produced on C.mmp in Bliss11. It uses the microcode to make the 11/40E emulate the computer module (LSI-11 with Slocal), and runs

under Hydra as user process(es).

The StarOS operating system was constructed and demonstrated by May 1978. It ran on a 10 processor, single-cluster configuration. StarOS is a capability based operating system building on research from the FAMOS and Hydra projects. StarOS provides an OS kernal that supports module definition, environment specifications, process splitting and joining, and the grouping of processes (modules executing in parallel) into Task Forces. In addition, synchronization and communication are provided at many levels of abstraction (eg. locks, mailboxes).

Software facilities currently available are an Algol 68 runtime system and a hardware monitor system for reliability diagnostics.

6.5.8 Analysis of CM*

An initial cost/performance analysis of Cm* has been carried out (Fuller at al, 1978). The analysis was made using a simple version of kmap microcode that provides basic communication and memory sharing functions. Five programs (eg. PDE, Sorting, Integer programming, Harpy, Algol 68) were used to analyse Cm*. Measurements of Memory Reference times (3.5 us local, 9.3 us intracluster, 26 us intercluster), local hit ratios (82-99%), execution speedup, and Bus contention were made. It was found that under the condition of a 90% local hit ratio to local memory, an almost linear speedup can be achieved in some applications. In addition, hierarchial switching structures as used in Cm* can provide good performance.

The blackboard control structure from Hearsay II was not transfered to Cm*. This is due to the inavailability of L* on Cm* and the de-emphasis placed on the task within the environment.

6.5.9 Alternative systems to permit evaluation

We are committed to exploring the space of multiprocessor systems, but we also believe that only by exploring the alternatives to our main systems can we really discover what is critical. Two such explorations were carried out.

A network is the major alternative structure to that of a multiprocessor system. (A network exchanges messages with the nodes standing as peripheral systems to each other; a multiprocessor shares address spaces.) Networks are the dominant form of multiple computer system under study today (though not necessarily for the right reasons). A major open question is whether the close coupling permitted in multiprocessing structures is worth having.

Using the Cm* hardware, both the Cm* multiprocessor system and a local network were analysed and compared (Raskin, 1978). Two types of measurements were made: 1) Benchmark programs and 2) Performance models (verified by measurements).

For the multiprocessor system (Cm*), a complex multi-cluster, queuing network model with several classes of customers was developed, analysed and validated at the PMS level. Performance was shown to be a function of the relative access times to modules within the cluster hierarchy. Other insights were realized through model manipulation.

For the local network system (CM-Net) a performance model was constructed. Performance was found to be a function of:

1. Message rate between processors.
2. Network transport communication rate.
3. Number of processors.

Comparing the two sysems, multiprocessor systems were found to be better for the applications studied (cooperative problem solving). Local network structures suffice under the condition of low communication rates. In either case, it is possible to classify application as to which system structure is more applicable.

The second system, C.vmp, investigated was a triplicated module, fault-tolerant system constructed with LSI-11s (Siewiorek et al, 1978a, 978b). This work is supported under contract to DEC. Issues such as reliability/performance tradeoffs; effect on software systems; and

synchronization were investigated. C.vmp was constructed so that the processors can run either in voting mode or independently. Ths allows a hybrid usage of the system. The software available was a standard RT-11 operating system. Hence C.vmp was invisible to the user. The use of voting and its entailed synchronization results in a 14% degradation in execution. The reliability of the system is excellent. Transient errors, and even the removal of hardware (e.g., a processor, memory boards) do no affect the system.

6.6 Annotated Bibliography

This section contains a partial list of reports published in the area of Multiple Processor Systems. An abstract accompanies each.

Barbacci, M., Siewiorek, D. P., Gordon, R., Howbrigg, R. and Zuckerman, S. Architecture research facility: ISP descriptions, simulation, data collection. In Computer Family Architecture Selection Committee Final Report. (Smith, W., Ed.) Naval Research Laboratory, Washington, DC, 1976.

The objectives of this paper are twofold. In the first place we discuss some issues related to the formal description of computer systems and how these issues were handled in a specific project, the selection of a standard computer architecture for the Army/Navy Computer Family Architecture (CFA) project. The second purpose is to present a methodology for automatically gathering architectural data which can be used for evaluation and comparison purposes. We will not discuss the rationale behind the selection of specific test programs and the statistical experiment set up to ascertain the influence of the programmers, the test programs, and the machine architecture on the results. These issues and the actual results of the experiment belong in a companion paper. Formal descriptions of three candidate architectures (IBM S/370, Interdata 8/32 and DEC PDP-11) were written in ISP, a computer description language. The ISP descriptions of the three architectures were used to simulate the execution of assembly language test programs. The measurements collected during the program simulations were stored into a data base for post-processing. Automating the data collection process not only eliminated tedious and potentially error prone hand calculations, but also provided the means to gather information about dynamic program behavior, information that would be almost impossible to calculate manually.

Barbacci, M. and Siewiorek, D. P. Evaluation of the CFA test programs via formal computer descriptions. COMPUTER 10, 10 (October 1977).

This article describes an architectural research facility used for the comparison of computer architectures. The facility consists of a compiler, simulator, symbolic translator, and data analysis programs. A formal computer description language, ISPL, is used to define the architecture of a computer. The description is compiled into an intermediate form that can be linked to, and interpreted by, a simulator. A symbolic translator

converts assembler-generated outputs of test programs into commands that initialize the simulator. After execution of the test program, a post-processor program summarizes the dynamic behavior. We believe this is the first time the architectures of commercially viable computers have been described in a formal language, descriptions compiled, and then used to drive a simulator, executing benchmarks and diagnostic machine language programs. This architectural research facility was used to collect data on dynamic performance of the three Computer Family Architecture candidates. A comparison article describes the test programs and three implementation-independent parameters--S (static measure of amount of program storage), M (dynamic measure of memory activity), and R (dynamic measure of register activity). Without the ISPL facility, collection of the dynamic M and R parameters for the candidate architectures would have been a tedious, error-prone task.

Barbacci, M. and Parker, A. The use of emulation in the verification of formal architecture descriptions. COMPUTER 11, 5 (May 1978).

Formal descriptions of digital systems have been used mainly for academic purposes. A recently completed project, however, illustrates that such descriptions are highly useful in the evaluation of commercially available computer architectures. A formal description can serve as the specification of a family of architectures. Such a specification, to be of value, must accurately reflect the actual architecture. Verification of this relationship, therefore, is central to the description's usefulness as a specification. Once verified, the description can generate machine- and implementation-independent diagnostics to be used in the evaluation of any architecture claiming family membership. The Computer Family Architecture Project evaluated a set of commercially available architectures in order to select a standard family of tactical military computers. Three of these architectures (IBM S/370, Interdata 8/32, and DEC PDP-11) were selected for a detailed comparison via test programs executed under an instrumented simulator. The results of the experiment, together with an evaluation of the existing software bases, were used to select the PDP-11 as the preferred architecture. A simulator and associated tools were used to verify the description of the selected architecture. Although the PDP-11 is the only architecture used in this article, the method is general and can be applied to other (not necessarily existing) computer architectures. Since formal descriptions have been used most for academic purposes, the use of ISP in the CFA Project was, to our knowledge, the first time descriptions of real, commercially available architectures were used for

evaluation and verification. Other projects show a range of applications of formal descriptions well beyond what will be described in this article.

Fuller, S. H., Jones, A. K. and Durham, I. Cm* Review. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1977.

The Cm* project has been in progress almost exactly two years. The present review is intended to critically examine our progress to date and evaluate our plans for the future of Cm*. We will not review the architecture, hardware implementation, or oprating system designs in this report. The Cm* hardware and software designs were described in a session at Ncc'77 [1,2,3]. This report contains a description of the measurement and evaluation studies conducted in the spring of 1977. Some of the data included here is peliminary. However, given the dearth of quantitative knowledge about multiple processor systems, we wanted to evaluate Cm* as early in its development cycle as possible to detect and correct flaws in our design and understanding of this exprimental multiprocessor. We have made an effort to describe the present state of Cm* and tabulate all of our current measurements so a dilligent reader can study and evaluate for himself our recent results.

Fuller, S. H., Ousterhout, J., Raskin, L., Rubinfeld, P., Sindhu, P. and Swan, R. Multi-microprocessors: An overview and working example. Proceedings of the IEEE, , February, 1978.

Rapid and continuing advances in large-scale integrated (LSI) semiconductor technology have lead to considerable speculation on ways to exploit microprocessors for building computer systems. Microprocessors are being applied very successfully where small amounts of computing power are needed, such as in calculators, instruments, controllers, intelligent terminals, and more recently in consumer goods and games; but it remains an open problem to design a commercially viable multiple-microprocessor structure. A variety of organizations have been proposed for such systems, and this article begins with an overview of this spectrum. Few multiple-microprocessor systems, however, have been built or otherwise subjected to a criticial analysis. To address the unresolved problems facing such systems, Carnegie- Mellon University has undertaken the design, implementation, and evaluation of an experimental multi-microprocessor computer system called Cm* Many of the

names and abbreviations of computer system components used here are derived from the PMS notation [4]. PMS is a scheme for concisely representing the "block diagram" level of computer organizations. Common abbreviations in PMS are P for processor, M for memory, S for switch, K for a control unit, C for computer, and L for a communications link. Suffixes are attached to provide more detailed information. For example, Pc for central processor, Pio for I/O processor, Mp for primary memory, and Ms for secondary memory. The name Cm* stands for an arbitrary number of Cm's, or Computer Modules, where the * is derived from the notation introduced by Kleene for regular expressions. Cm* is pronounced see-em-star.). A 10-processor, 1/2 Mbyte primary memory prototype configuration of Cm* has been completed and became available for experimentation in the Spring of 1977. The kernel of an operating system and five application programs with widely varying characteristics have been written for Cm*, and form the basis for the measurements and discussion given here. Several of the application programs have been able to utilize all the processors in the prototype system effectively. In other words, doubling the number of available processors effectivel doubled the execution speed of these programs.

Fuller, S. H. and McGehearty, P. F. Minimizing latency in CCD memories. IEEE Trans. on Computers, Vol. C-27, March, 1978.

Serial memories built from charge-coupled devices (CCD's) offer an opportunity for minimizing latency times not available with the more conventional drum and disk (serial) memory units. Let r be the ratio of the maximum to the minimum clocking rates for the CCD memory. We show that in many practical situations the average latency can be reduced from 1/2 to 1/(1 plus sq. rt. of r) of a revolution time if the optimal clocking strategy is used when the CCD is idle.

Ingle, A. D. and Siewiorek, D. P. Reliability modeling of multiprocessor structures. Proceedings of CompCon, Washington, DC, September, 1976. Also CMU-CSD Technical Report September, 1976.

Multiprocessor systems, although designed for speed and processing power, lend inherently to redundancy. Appropriately designed distributed intelligence systems that utilize system reconfiguration and graceful degradation can be substantially more reliable than uniprocessor systems. Reliability models for two multiprocessor systems, C.mmp and Cm*, are presented and compared to a single LSI-11 processor. With the exception of spacebourne systems, most systems may be subjected to

tests to ensure proper functioning. When performed regularly, these integrity checks enhance confidence in the system, and its expected mean time to failure. Effect of such periodic maintenance is modeled. The expected life is seen to depend strongly on the efficiency of the tests. The improvement in expected life, however, is observed to be limited by non-redundant parts of a system. Under periodic maintenance, Cm* system offers greater life than C.mmp for tasks allowing considerable redundancy.

Jones, A. K., Chansler, R. J., Jr., Durham, I., Feiler, P. and Schwans, K. Software management of Cm*, a distributed multiprocessor. Proceedings of the National Computer Conference, Dallas, TX, June, 1977.

This paper describes the software system being developed for Cm*, a distributed multi-microprocessor. This software provides for flexible, yet controlled, sharing of code and data via a capability addressed virtual memory, creation and management of groups of processes known as task forces, and efficient interprocess communication. Both the software and hardware are currently under construction at Carnegie-Mellon University.

Newell, A. and Robertson, G. C.mmp: A progress report on synergistic research. In Perspectives on Computer Science. (Jones, A. K., Ed.) Academic Press, New York, NY, 1977, pp. 147-182.

C.mmp has become a synergistic research effort in which almost ten distinct research efforts have become intertwined, so that they both support each other and depend on each other in ways that are both gratifying and scare. These efforts include: the hardware effort of C.mmp; the operating system, HYDRA; the implementation language, BLISS-11; the hardware performance monitor; the driving application, HEARSAY-II (a speech-understanding system); a second implementation system, L*; the development of a dynamically microprogrammed PDP-11/40; the development of specialized processors; and the problem of decomposition of algorithms into parallel form. The end is not in sight, in fact. We attempt here an integrated picture of this total research endeavor, making clear what the iimportant scientific issues are and relating these to the current state of progress. Our interest in such an endeavor includes its role as a model for how experimental computer science can be accomplished. We reflect on this as well.

Oleinick, P. and Fuller, S. H. The implementation and evaluation of a parallel algorithm on C.mmp. Technical Report, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, June, 1978.

C.mmp is a multi(mini) processor with up to sixteen processors. This paper presents and discusses measurements of the C.mmp system at several levels: Basic hardware performance measurements, runtime performance of Hydra, C.mmp's operating system, and overall performance of a particular application: a parallel rootfinding algorithm. The purpose of this paper is to get a detailed look at the performance of an implementation of a parallel program on C.mmp. The rootfinding algorithm was chosen because it meets two constraints: it is a parallel algorithm with significant interprocess communication; and it is of relatively low complexity, enabling us to focus on implementaton issues rather than subtleties in the algorithm itself. Variations in processor speeds and asynchronously executing operating system functions are shown to have a detrimental effect on the rootfinder's performance. However, the most important implementation decision affecting the performance of the rootfinding program is the type of synchronization semaphore used. We define the threshold for practical application of a semaphore to be when 50 per cent of the execution time is attributed to semaphore related overheads. Using the 50 per cent criteria, we measured thresholds for inter-synchronization times from two milliseconds for the most primitive locks, to 200 milliseconds for the most sophisticated and flexible semaphore.. During the course of these measurements, Hydra underwent several revisions and the 200 millisecond threshold was reduced to 33 milliseconds. The principal concept responsible for this performance improvement is discussed in the paper.

Siewiorek, D. P., Canepa, M. and Clark, S. The architecture and implementation of a fault-tolerant multiprocessor. Proceedings of the International Symposium on Fault-Tolerant Computing, Los Angeles, California, June, 1977. Also CMU-CSD Technical Report December, 1976.

The architecture of a multiprocessor with a fault tolerant operating mode is described and analyzed. A bus level voter is used to satisfy the stringent design constraints of software transparency (programs from non-redundant versions will execute in a fault tolerant manner without modification), modularity, use of off-the-shelf components, and dynamic trading of performance for reliability. Bus level voting also allows handling of diverse system components (processors, memories, floppy disks, teletypes,

etc.) in a uniform way. Models of performance degradation (20 per cent slower than non-redundant on instruction execution rate, 50 per cent slower on expected disk latency) and reliability improvement (both permanent and transient failures) are presented as well as experience in redundant system debugging, system initialization and switchover software, and initial performance measurements. The system, which is nearing completion, will be used to measure the occurrence of transient failures and to test fault tolerant bus protocols.

Swan, R. J., Fuller, S. H. and Siewiorek, D. P. Cm*: A modular, multi-processor. Proceedings of the National Computer Conference, Dallas, TX, June, 1977.

This paper describes the architecture of a new large multiprocessor computer system being built at Carnegie-Mellon University. The system allows close cooperation between large numbers of inexpensive processors. All processors share access to a single virtual memory address space. There are no arbitrary limits on the number of processors, amount of memory or communication bandwidth in the system. Considerable support is provided for low level operating system primitives and inter-process communication.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-79-0043	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): RESEARCH IN INFORMATION PROCESSING	5. TYPE OF REPORT & PERIOD COVERED: Final
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Allen Newell, Joseph Traub	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Department of Computer Scienct, Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101E, A02466/7
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, Virginia 22209	12. REPORT DATE: December 1978
	13. NUMBER OF PAGES: 115
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

119p. Final rept. 1 Jul 76 - 30 Jun 78,

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the final report on Research on Information Processing, supported by Defense Advanced Research Projects Agnecy and monitored by the Air Force Office of Scientific Research under contract number F44620-73-C-0074. It covers the contract period July 1, 1976 to June 30, 1978.

The report is organized as five chapters under five major areas of research: Image Understanding systems, Speech Understanding systems, Machine

DD FORM 1 JAN 73 1473

403 081

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract continued.

Intelligence, Software Technology, and Multiprocessors. Each chapter is self-contained and can be read without reading the entire report.

Carnegie-Mellon University
Department of Computer Science
Schenley Park
Pittsburgh, Pennsylvania 15213